UNIVERSIDAD DE LA RIOJA

TRABAJO FIN DE ESTUDIOS

Título

Guiando la creación de modelos de detección de objetos basados en deep learning

Autor/es

ÁNGELA CASADO GARCÍA

Director/es

JÓNATAN HERAS VICENTE

Facultad

Escuela de Máster y Doctorado de la Universidad de La Rioja

Titulación

Máster Universitario en Tecnologías Informáticas

Departamento

MATEMÁTICAS Y COMPUTACIÓN

Curso académico

2017-18

publicaciones.unirioja.es
E-mail: publicaciones@unirioja.es

Trabajo de Fin de Máster

Guiando la creación de modelos de detección de objetos basados en *deep learning*

Autor:

Ángela Casado García

Tutor: Jónathan Heras Vicente

MÁSTER:
Máster en Tecnologías Informáticas (853M)

Escuela de Máster y Doctorado

AÑO ACADÉMICO: 2017/2018

Agradecer a Jónathan, su total dedicación, su gran paciencia (no sé como me has aguantado tanto ☺), gracias por tu valiosa ayuda, de no haber sido por ti este trabajo no hubiera sido tan chuli y por supuesto, sin ti no hubiera sido tan divertido y tan genial.
Dar las gracias a la empresa Pixelabs, que me ha brindado todas las herramientas necesarias para completar este trabajo.
También a mis amigos, que siempre han estado y siguen estando ahí para apoyarme en los momentos difíciles y han conseguido animarme cuando me hacía falta, Jorge va por ti y tu aguante.
Y por último, a mi estupenda familia, por estar siempre conmigo, saber que puedo contar con ella en todo momento, es un enorme apoyo, son mi esencia para seguir adelante. En especial a mi madre.

Índice

Resumen

El problema de la detección de objetos en imágenes lleva años presente dentro de la visión por computador. La detección consiste en encontrar y clasificar una cantidad variable de objetos en una imagen. En las últimas décadas se han investigado distintos enfoques para encontrar soluciones óptimas a este problema, entre las que destacan las técnicas de *deep learning* que nos ofrecen grandes avances en la actualidad. El *deep learning* permite enseñar a los ordenadores el aspecto que poseen diferentes objetos para identificar posteriormente objetos similares. Aunque estas técnicas son muy exitosas son difíciles de usar.

YOLO es una de las técnicas de deep learning más utilizadas en la detección de objetos en imágenes. YOLO permite la construcción de modelos precisos que se pueden emplear en aplicaciones en tiempo real. Sin embargo, como la mayoría de las técnicas de *deep learning*, YOLO tiene una curva de aprendizaje abrupta y la creación de modelos utilizando este enfoque puede ser un desafío para los usuarios no expertos.

El objetivo de este trabajo ha consistido en desarrollar un asistente que guíe la creación de modelos de detección basados en YOLO para usuarios expertos y no expertos en el tema. Para desarrollar dicho asistente se han usado los cuadernos de Jupyter, que nos permiten interactuar con código Python intercalado con texto explicativo de manera sencilla. De este modo, cualquier usuario puede construir de manera rápida y sencilla sus propios modelos de detección.

Para construir dicha herramienta ha sido necesario aprender y aplicar diversos conceptos sobre aprendizaje automático, *deep learning*, análisis de imagen y detección de objetos.

Abstract

The problem of object detection has been present for years in the computer vision field. Detecting objects in an image consists in locating and classifying a variable amount of objects in an image. In the last decades, different approaches have been studied to find optimal solutions for this problem; and, among them, we can highlight deep learning techniques, that, nowadays, offer great advances. Deep learning techniques allow us to teach computers the aspect of different objects in order to identify similar objects in the future. Even if deep learning techniques are successful, they are difficult to use.

YOLO is one of the most employed deep learning approaches in object detection. YOLO allows us to build accurate models that can be employed in real-time applications. However, as most deep learning techniques, YOLO has a steep learning curve, and creating models using this approach might be a challenge for non-expert users.

The goal of this work has consisted in developing an assistant that guides both expert and non-expert users in the creation of detection models using YOLO. In order to develop such an assistant, we have employed Jupyter notebooks that introduce explanations in a simple way, and allow users to interact with Python code. In this way, users can build, in a simple and fast way, their own detection models.

In order to build the assistant, it has been necessary to learn and apply several concepts about machine learning, deep learning, image processing and object detection.

1. Introducción

En esta sección se van a detallar los antecedentes de la empresa y el grupo de investigación donde se realiza el trabajo, además se explican cuales son los problemas que han motivado la realización de este proyecto, y el objetivo que se intenta lograr.

1.1. Antecedentes

Este proyecto se enmarca dentro del trabajo realizado por la empresa Pixelabs y el grupo de Informática de la Universidad de la Rioja. Pixelabs surge de la iniciativa privada de tres entusiastas de la innovación y de las nuevas tecnologías. Desde principios de 2015 incuban la idea de aplicar técnicas de reconocimiento facial y de patrones al campo de la publicidad, en concreto, al reconocimiento de identidad corporativa (logotipo, marca, imagen corporativa) en entornos audiovisuales.

A día de hoy, los anunciantes y marcas no tienen una herramienta de medición precisa que les permita conocer y evaluar el impacto de sus campañas en medios audiovisuales. Solo se mide la duración de un promocional y la audiencia potencial, pero en ningún caso, se cuantifica la presencia de los logotipos u otros elementos diferenciales de referencia de una marca durante la retransmisión de un evento. La idea de Pixelabs es aparentemente sencilla y consiste en capturar en tiempo real la emisión de un evento audiovisual y ser capaces, mediante la tecnología apropiada, de reconocer (y cuantificar) los logotipos de determinadas marcas que aparezcan durante la emisión del evento.

Esta idea permitió desarrollar un proyecto de reconocimiento de logos, gracias al cual han ido surgiendo otros nuevos proyectos de detección de objetos en vídeos.

Por su parte, el grupo de Informática del Departamento de Matemáticas y Computación de la Universidad de La Rioja (a partir de ahora ginfor) trabaja desde 2011 en proyectos de análisis de imágenes biomédicas. En concreto se han abordado problemas relacionados con el análisis de imágenes de neuronas en enfermedades neurodegenerativas [1], de geles de ADN [2] o de susceptibilidad de bacterias [3].

1.2. Justificación del proyecto

“Detectar logos en vídeos”, “detectar qué obreros llevan casco y cuáles no en una construcción”, “detectar señales de tráfico”, “detectar las sinapsis de una neurona”, “detectar los estomas de una imagen de una planta”, “detectar...”; todos estos son proyectos de la empresa Pixelabs o del grupo ginfor y todos empiezan con la palabra clave “detectar”; distinto tema, distintos lugares, pero al final, todos los proyectos siguen el mismo proceso:

1. Generar el banco de imágenes o vídeos.
2. Indicar dónde se encuentra el banco de imágenes o vídeos.

3. Integrar una herramienta que nos permita anotar cada imagen o frame del vídeo y guardar las imágenes anotadas en un formato predeterminado.
4. Lanzar el entrenamiento de una red neuronal para generar un modelo de detección adecuado al problema.
5. Usar una aplicación que dadas nuevas imágenes o vídeos y el modelo entrenado, nos permita obtener las predicciones.
6. Generar distintas estadísticas a partir de las predicciones obtenidas como resultado del paso anterior.

Pero este proceso cuenta con varios problemas:

- Consta de muchos pasos, donde cada uno de ellos depende del anterior, lo cual exige seguir un orden, de manera que mientras no termine el primero, no puede empezar el segundo y así sucesivamente.
- Cada paso lo realiza una persona especializada y preparada para él únicamente, no siendo factible que esa persona tenga conocimientos específicos del resto de pasos. Puesto que cada uno de ellos tiene muchas y diferentes configuraciones.
- Dificultad a la hora de realizar prototipos rápidos, debido a que la creación de un modelo demanda excesivo tiempo.
- El proceso no se puede realizar en cualquier tipo de ordenador porque se requiere de unas características técnicas mínimas.

El objetivo de este proyecto, consiste en realizar un aplicativo que englobe todas las tareas de estos proyectos y que una única persona sea capaz de llevar a cabo todo el desarrollo, sin que sea necesario que esté especializada en todas las fases. En concreto, lo que se pretende conseguir es realizar un asistente que nos vaya indicando los pasos necesarios para realizar la detección automática de algún objeto en vídeos o imágenes, sin tener que cambiar continuamente de aplicación, permitiendo de esta manera que los nuevos proyectos que puedan surgir se puedan abordar de manera más rápida y sencilla.

Cuando pensamos en un asistente nos viene a la cabeza una herramienta que oculta mucha información y que el usuario utiliza sin necesidad de aprender los conceptos subyacentes. Nuestro asistente no solo servirá para crear modelos de detección en imágenes y vídeo sino también permitirá aprender que está sucediendo por debajo. Para ello se utilizarán los notebooks de Jupyter [4], que son documentos para publicar código, resultados y explicaciones en una forma legible y ejecutable. El conjunto de notebooks se puede descargar desde la página de github https://github.com/ancasag/YOLONotebooks y en el anexo se puede ver una lista de estos notebooks.

Este trabajo va a estar estructurado en cinco bloques, donde iremos viendo los conceptos previos necesarios para su entendimiento, el funcionamiento de las técnicas que se van a usar, los pasos necesarios para su uso, se realizarán casos de estudio poniendo en práctica las técnicas estudiadas y se terminará con las conclusiones obtenidas.

2. Conceptos previos

En esta sección vamos a ver los conceptos previos necesarios para el entendimiento y desarrollo del proyecto.

2.1. Problemas de visión por computador

La visión es uno de los sentidos más importantes de los seres humanos. Tanto es así que se calcula que más del 70% de las tareas del cerebro son empleadas en el análisis de la información visual [5]. Casi todas las disciplinas científicas hacen uso de la visión para llevar a cabo sus estudios, es una actividad inconsciente y aún hoy, es muy complicado saber cómo se produce con exactitud. Ahora que tecnologías como el *deep learning* están en auge, ha llegado también una nueva revolución, la visión por computador, aunque todavía queda mucho camino por recorrer.

La visión por computador es una disciplina de la inteligencia artificial, y aunque la inteligencia artificial surgió en los años 50 del siglo pasado, grandes empresas apuntan que apenas estamos viviendo la primera edad de la Inteligencia Artificial [6], ya que las máquinas comienzan a realizar procesos de aprendizaje del habla y el reconocimiento de imágenes al mismo nivel que las personas.

En la visión por computador podemos distinguir 4 grandes problemas a resolver: la clasificación, la localización, la detección y la segmentación. Como ya hemos comentado, en este trabajo estamos interesados en el problema de la detección, pero es necesario contextualizar también los otros problemas.

- ***Clasificación***. Probablemente es el problema más conocido en la visión por computador y consiste en clasificar una imagen en una de las muchas categorías posibles. En los últimos años, los modelos de clasificación han superado el rendimiento humano y el problema de la clasificación se considera prácticamente resuelto, como en *GoogleNet* [7] o *ResNet [8]*. Un ejemplo del problema de la clasificación puede verse en la figura 1.

Figura 1. Clasificación de un objeto.

- ***Localización***. La localización encuentra la ubicación de un solo objeto dentro de la imagen. La localización puede usarse para resolver muchos problemas útiles de la vida real, por ejemplo, el recorte inteligente [9] (saber cómo recortar imágenes en función de dónde se encuentra el objeto), o incluso la extracción regular de objetos para su posterior procesamiento utilizando diferentes técnicas [10]. Se puede combinar con la clasificación para localizar no solo el objeto, sino también para categorizarlo en una de las muchas categorías posibles. Podemos ver un ejemplo en la figura 2.

Figura 2. Localización de un objeto.

- ***Detección***. Al iterar sobre el problema de localización más clasificación, terminamos con la necesidad de detectar y clasificar múltiples objetos al mismo tiempo. La detección de objetos es el problema de encontrar y clasificar una cantidad variable de objetos en una imagen. La diferencia importante es la parte "variable". A diferencia de problemas como la clasificación, la salida de la detección de objetos es de longitud variable, ya que la cantidad de objetos detectados puede cambiar de una imagen a otra, ver figura 3.

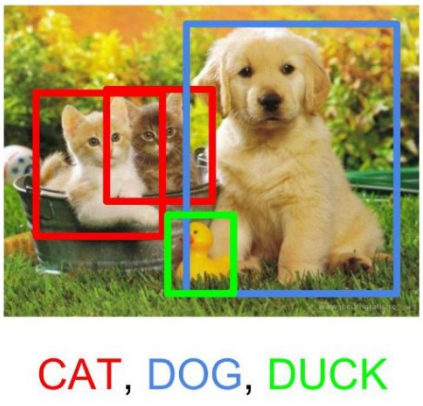


Figura 3. Detección de varios objetos en una imagen.

- ***Segmentación***. Yendo un paso más allá de la detección de objetos, no solo queremos encontrar objetos dentro de una imagen, sino también encontrar una máscara de píxel por píxel de cada uno de los objetos detectados. Nos referimos a este problema como instancia o segmentación de objetos, un ejemplo puede verse en la figura 4.

Figura 4. Segmentación de una imagen.

Dentro de la visión por computador, la detección de objetos es uno de los temas más candentes. El problema parece sencillo: dada una imagen queremos ser capaces de encontrar en ella varios objetos, como una silla, un libro o un ordenador. Para una persona esta tarea es algo obvio pero para una máquina no lo es en absoluto. Para entender el problema hay que pensar en cómo queda codificada una imagen digital. En general, para una máquina las imágenes son enormes cajas tridimensionales (matrices) llenas de números. De manera habitual, cada píxel o punto de la imagen queda representado con tres valores, que codifican su color como una combinación de rojo, verde y azul. Así pues, cuando una máquina busca un objeto dentro de una imagen lo que realmente hace es buscar patrones que se correspondan con el objeto en particular dentro de las matrices. Un ejemplo de las matrices y como cada píxel queda representado con tres valores se puede ver en la figura 5.

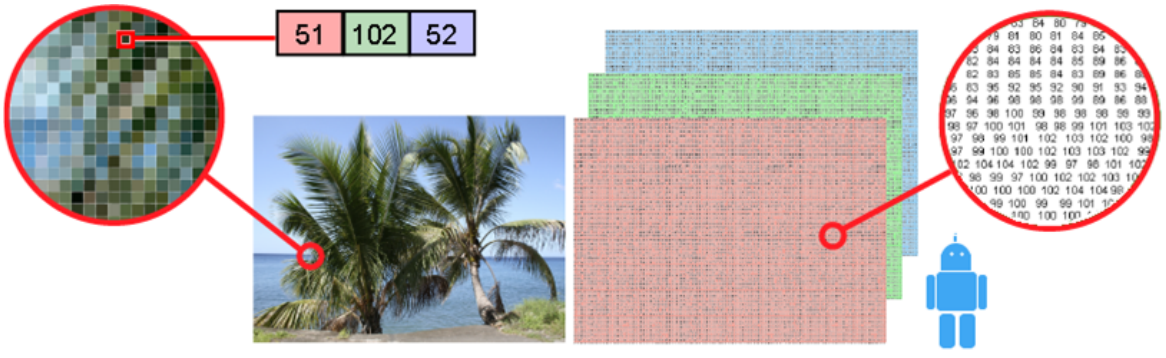


Figura 5. Representación de una imagen.

Después de esta breve introducción a las distintas tareas de la visión por computador, vamos a ver cómo abordaremos en este TFM el problema de la detección. En concreto,

vamos a ver la detección de objetos como un problema de aprendizaje automático, en el cual utilizaremos técnicas de *deep learning* que explicamos a continuación.

2.2. Aprendizaje automático

El aprendizaje automático [11,12] (en inglés *machine learning*) es el diseño y estudio de las herramientas informáticas que utilizan la experiencia pasada para tomar decisiones futuras, es decir, es el estudio de programas que pueden aprender de los datos. El objetivo fundamental del aprendizaje automático es generalizar, o inducir una regla desconocida a partir de ejemplos donde esa regla es aplicada. El aprendizaje automático combina conceptos y técnicas de diferentes áreas del conocimiento, como las matemáticas, la estadística y las ciencias de la computación; por tal motivo, hay muchas maneras de estudiar la disciplina.

El aprendizaje automático tiene una amplia gama de aplicaciones, incluyendo motores de búsqueda, diagnósticos médicos, detección de fraude en el uso de tarjetas de crédito, análisis del mercado de valores, clasificación de secuencias de ADN, reconocimiento del habla y del lenguaje escrito, juegos y robótica [13]. Pero para poder abordar cada uno de estos temas, es crucial, en primer lugar distinguir los distintos tipos de problemas de aprendizaje automático con los que nos podemos encontrar: aprendizaje supervisado, aprendizaje no supervisado, aprendizaje semi-supervisado, aprendizaje por refuerzo, transducción y aprendizaje multitarea [14,15].

Para este proyecto en concreto solo se hablará del aprendizaje supervisado, para más información del resto de tipos de aprendizaje automático consultar los anexos. En los problemas de aprendizaje supervisado, se enseña o entrena al algoritmo a partir de datos que ya vienen etiquetados con la respuesta correcta. Cuanto más grande sea el conjunto de datos, más aprenderá el algoritmo sobre el tema. Una vez concluido el entrenamiento, se le brindan nuevos datos al algoritmo, pero sin las etiquetas de las respuestas correctas, y el algoritmo de aprendizaje utiliza la experiencia pasada que adquirió durante la etapa de entrenamiento para predecir un resultado. Existen diversos algoritmos de aprendizaje supervisado, por ejemplo K vecinos más cercanos (KNN) [16], *random forest* [17], máquinas de vectores de soporte (SVM) [18] o redes neuronales artificiales (MLP) [19].

De acuerdo con el *No Free Lunch Theorem* [20] no existe el mejor modelo de aprendizaje supervisado para resolver todos los problemas, y por lo tanto es necesario probar distintos algoritmos para resolver cada problema concreto.

Uno de los algoritmos que más relevancia ha adquirido en los últimos años son las redes neuronales artificiales. Las redes neuronales artificiales (surgidas en los 50) son una clase de algoritmos de aprendizaje automático que aprenden de datos y se especializan en reconocimiento de patrones, inspirados por la estructura y función del cerebro. El primer modelo de red neuronal fue propuesto en 1943 por McCulloch y Pitts, este modelo era un

modelo binario [21]. A finales de 1950, Frank Rosenblatt y otros investigadores desarrollaron una clase de redes neuronales llamadas perceptrones [22], que sirvieron para resolver problemas de reconocimiento de patrones. En los últimos años las redes neuronales están teniendo un gran éxito debido a:

- La gran cantidad de datos.
- El aumento de la capacidad de procesamiento usando GPUs.
- La mejora de algoritmos de entrenamiento.

De las redes neuronales ha surgido una nueva disciplina conocida como *deep learning*.

2.3. Deep learning

El *deep learning* (o aprendizaje profundo) es sin duda el área de investigación más popular actualmente dentro del campo de la inteligencia artificial [23]. La mayoría de las nuevas investigaciones que se realizan trabajan con modelos basados en técnicas de *deep learning*; ya que las mismas han logrado resultados sorprendentes en campos como el procesamiento del lenguaje natural [24] y la visión por computador [25]. El *deep learning* es un subcampo del aprendizaje automático (ver figura 6) y a veces los términos se confunden por lo que explicamos a continuación la diferencia entre ellos.

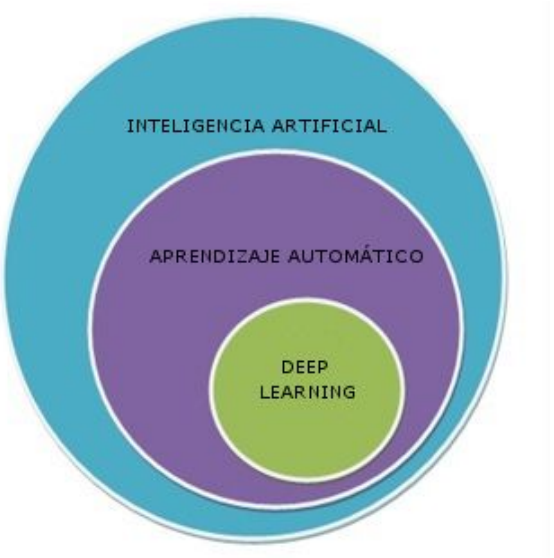


Figura 6. Diagrama de Venn que muestra la relación entre deep learning, aprendizaje automático e inteligencia artificial.

Centrémonos en el caso de aprendizaje supervisado tradicional para la visión por computador. En este área, es una persona la que tiene que seleccionar unos descriptores o diseñar un programa que extraiga descriptores (por ejemplo tenemos LBP [26] o Haar [27] que sirven para determinar las características relevantes de una imagen). En concreto, el objetivo es crear unos descriptores lo suficientemente discriminativos como para describir el contenido de una imagen. Pero tenemos que tener en cuenta que puede haber problemas como la gran variabilidad, la iluminación, las oclusiones, los cambios de ángulos o los fenómenos atmosféricos, que nos pueden producir variaciones en la manera de percibir la

imagen. Esta es la tarea más difícil que nos podemos encontrar en el aprendizaje supervisado tradicional ya que en caso de no obtener unos buenos descriptores lo más seguro es que el algoritmo de aprendizaje sea incapaz de funcionar correctamente.

El *deep learning* intenta resolver este tipo de problemas, ya que se encarga de aprender de manera automática la representación de los datos en términos de otras representaciones más sencillas gracias a una jerarquía de creciente complejidad y abstracción, ver figura 7.

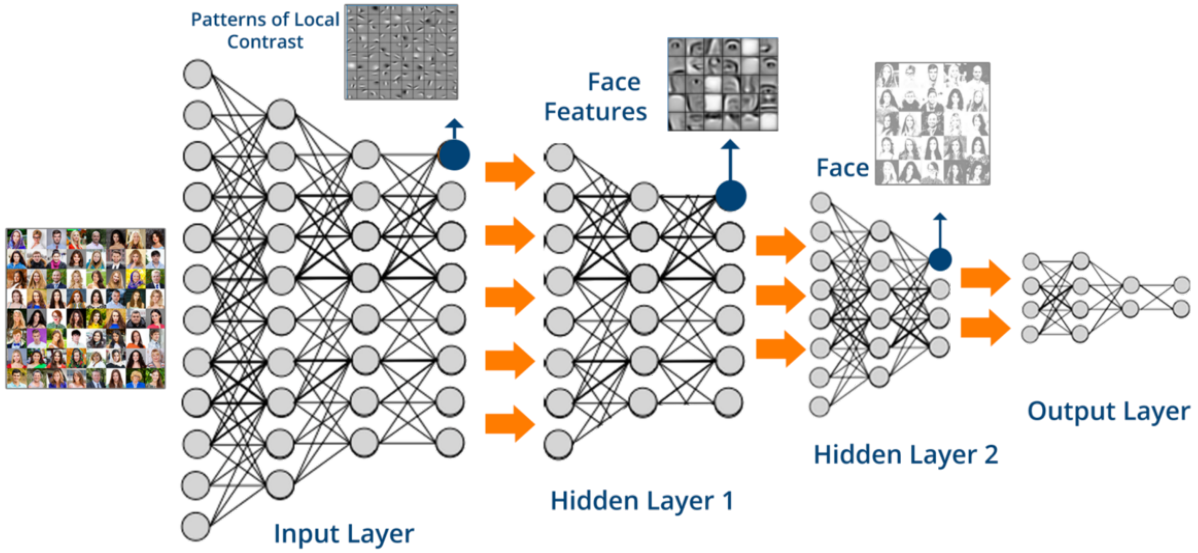


Figura 7. Representación deep learning.

Este aprendizaje de la representación se realiza en el proceso de entrenamiento donde las capas más bajas codifican una representación básica del problema y las capas más altas utilizan las capas bajas para construir abstracciones. Por ejemplo, en la figura 7, la primera capa se centra en patrones básicos, dichos patrones se utilizan para construir abstracciones como orejas, ojos, en capas cada vez más profundas que a su vez se utilizan para la construcción de las caras completas. Por lo tanto, en el aprendizaje automático tradicional se seleccionan de manera manual los descriptores de las imágenes, mientras que en el *deep learning* dichos descriptores son aprendidos a medida que se entrenan los algoritmos.

Una vez comentada la diferencia entre aprendizaje automático tradicional y *deep learning*, a continuación explicamos los bloques básicos del *deep learning* para la visión por computador, que son, principalmente, distintas arquitecturas de redes neuronales. Aunque existen muchos tipos de arquitecturas [28], las que a nosotros nos interesan son las *feed forward* y las convolucionales.

2.3.1 Redes neuronales tradicionales

En la base del *deep learning* se encuentran las redes neuronales clásicas también conocidas como *feed forward* (o *multilayer perceptron*) [29]. Para explicar el funcionamiento de una red neuronal clásica nos basaremos en la figura 8.

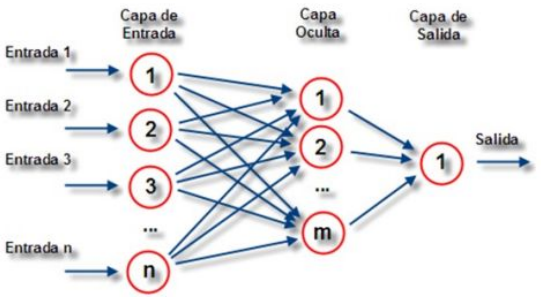


Figura 8 . Red neuronal feed forward.

En las redes neuronales *feed forward*, las neuronas se organizan en capas, donde cada una se conecta con todas las neuronas de la capa siguiente. Cada conexión tiene asociado un peso, por lo que, la principal operación que se realiza es una multiplicación entre el valor de la neurona y su conexión saliente. Las neuronas de las capas siguientes reciben los resultados anteriores sumados en uno y aplican una función no lineal para producir un nuevo resultado que se pasa a las neuronas de la siguiente capa. Existen numerosas funciones que podemos usar, pero cabe destacar la función sigmoide, que fue la base de la mayoría de las redes neuronales durante muchas décadas, aunque en estos últimos años ha perdido popularidad. En su lugar, la mayoría de las redes neuronales actuales usan otro tipo de función de activación llamada *rectified linear unit* o ReLU, que se define como R(z)=max(0,z) [30]. Las funciones sigmoide y ReLU están representadas en la figura 9.

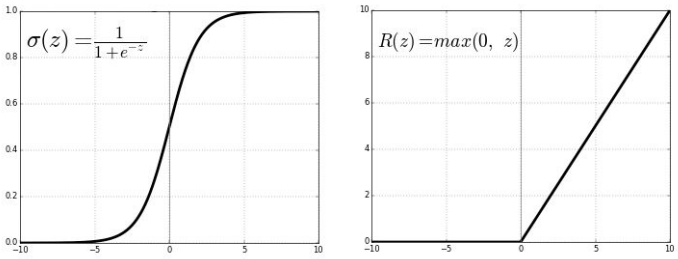


Figura 9 . Función sigmoide (izquierda) y función ReLU (derecha).

El entrenamiento de la red consiste en modificar la parte adaptable de la misma, que son los valores de los pesos, hasta conseguir el comportamiento deseado. Esto se lleva a cabo utilizando el algoritmo de *back propagation* que a partir de un conjunto de datos de entrenamiento calcula el error producido por la red e intenta minimizarlo modificando poco a poco los pesos. El proceso detallado puede consultarse en [31].

2.3.2 Redes neuronales convolucionales

Las redes neuronales más utilizadas actualmente en visión por computador son las convolucionales. El objetivo de estas redes es aprender la jerarquía de representación de imágenes. Los fundamentos de las redes neuronales convolucionales se basan en el Neocognitron [32], introducido por Kunihiko Fukushima en 1980. Este modelo fue mejorado

por Yann LeCun en 1998 al introducir un método de aprendizaje basado en *back propagation* para poder entrenar el sistema correctamente [33]. Estas redes, se construyen apilando una serie de capas, que van transformando la imagen de entrada través de una serie de filtros que aplican un operador de convolución [34], y cuyos valores podemos inicializar aleatoriamente. Estos valores serán los que se actualicen durante el proceso de aprendizaje.

El operador de convolución tiene el efecto de filtrar la imagen de entrada con un núcleo previamente entrenado. Esto transforma las imágenes de tal manera que ciertas características (determinadas por la forma del núcleo) se vuelven más dominantes en la imagen de salida al tener estas un valor numérico más alto asignados a los píxeles que las representan. Estos núcleos tienen habilidades de procesamiento de imágenes específicas, como por ejemplo la detección de bordes que se puede realizar con núcleos que resaltan el gradiente en una dirección en particular. Sin embargo, los núcleos que son entrenados por una red neuronal convolucional generalmente son más complejos para poder extraer otras características más abstractas y no tan triviales.

Como se ha comentado en la Introducción,uno de los objetivos de este trabajo es hacer más accesibles diversos conceptos de visión por computador y *deep learning*. Con los filtros aparece nuestra primera contribución en este sentido. En concreto, se ha desarrollado un notebook de Python donde se define y explica el operador de convolución, mostramos diversos filtros y cómo se usan, además de una comparación con la librería de visión artificial OpenCV [35]. También se han incluido otros filtros que no se usan con la convolución, para ver el notebook visitar la página https://github.com/ancasag/YOLONotebooks.

Las redes convolucionales están formadas por cinco tipos de capas principalmente:

- La capa de entrada: es la capa que recibe las imágenes sin procesar.
- Las capas de convolución (CONV): qué son las capas que le dan el nombre a la red y las que transforman las imágenes aplicando los filtros.
- Las capas de activación: estas capas aplicarán la función ReLU u otras similares a la salida de la capa CONV. Estas capas se aplican ya que dan buenos resultados, permiten representar modelos no lineales y resultan muy sencillas de usar.
- Las capas de *pooling* (POOL): Tras la capa convolucional y la ReLU se usa un proceso llamado pooling. Este proceso, reduce la dimensionalidad del mapa de características y retiene la mayor parte de la información. Estas capas subdividen el mapa de características en regiones, sobre las que aplican una operación matemática de la que se obtiene un único valor. Las operaciones más habituales son la suma, la media y el máximo, aunque la más utilizada es la del máximo (ver figura 10). Este proceso produce varias mejoras:
    - Reduce el número de parámetros, lo cual permite agilizar el proceso de entrenamiento.

- Consigue que la red sea invariante ante pequeñas transformaciones de traslación o rotación, debido a que tomamos el máximo de una región.
- También al reducir el número de parámetros disminuye la posibilidad de que haya sobreajuste.

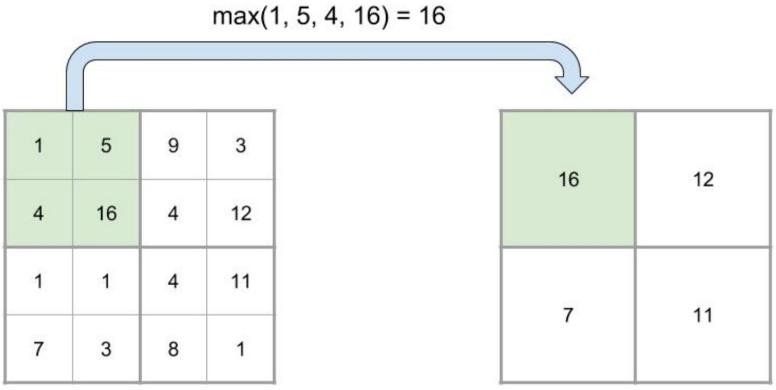


Figura 10. Agrupación por máximo, max-pool.

- Una capa clasificadora totalmente conectada (FC): Es una red neuronal clásica, *feed forward*, situada al final de la red, y es la encargada de dar el resultado final.

Existen diversas arquitecturas de redes convolucionales y su principal diferencia radica en el número y en la forma de combinar las capas de convolución, de activación y *pooling*. Algunas implementaciones específicas que podemos encontrar sobre este tipo de redes son: *Inception v3* [36], *ResNet* [8], *VGG16* [37] y *Xception* [38], entre otras. Todas ellas han logrado muy buenos resultados en el reto *ImageNet [39]* en el cual se evalúan los algoritmos para la clasificación de objetos a partir de imágenes.

Una arquitectura sencilla de visualizar se conoce como *Lenet5* y fue presentada por Yann Lecun en [33], ver figura 11.

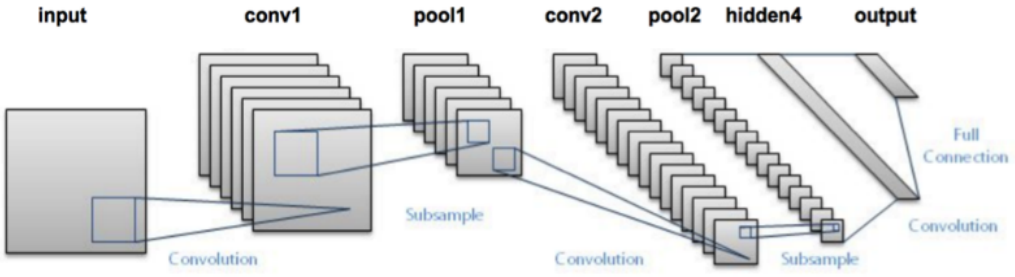


Figura 11. La arquitectura LeNet consiste en dos series de capas CONV -> ACTIV -> POOL seguidas de un feed forward.

Como hemos dicho existen diversas arquitecturas de redes convolucionales. Para comprender mejor su estructura se ha desarrollado un notebook de Python donde vamos a poder probar y trabajar con distintas redes convolucionales (por ejemplo, *GoogleNet* [7], *ResNet* [8], *VGG16* [37], y *LeNet* [33]). El problema que tienen las redes convolucionales es

que tienen un gran número de parámetros para entrenar. Estamos hablando de millones (por ejemplo *GoogleNet* [7] y *ResNet* [8]), la solución que se le aplica es el uso de una gran cantidad de datos y procesado de los mismos usando GPUs. Para mostrar la importancia de usar GPUs para entrenar este tipo de redes, se realiza una comparación de tiempos usando y sin usar GPUs en distintos entornos. También realizaremos una tabla comparativa de tiempos para la predicción. El notebook y las comparativas de tiempos pueden verse en https://github.com/ancasag/YOLONotebooks.

Una vez explicados diversos conceptos de aprendizaje automático y *deep learning* pasamos a ver cómo se aplican al problema concreto de este TFM: la detección de objetos.

2.4. El problema de la detección de objetos

La detección de objetos es un problema de visión por computador que, como hemos dicho anteriormente, consiste en localizar dentro de una imagen la posición de varios objetos indicando además la categoría a la que pertenecen dichos objetos.

Dependiendo del problema al que nos enfrentemos se pueden desarrollar soluciones *adhoc* que sirvan para abordarlo, pero esto no siempre es posible ya que depende de la variabilidad o complejidad de las imágenes y de los objetos a detectar.

En este apartado, empezamos a profundizar sobre cuáles son los principales problemas de la detección de objetos, el enfoque clásico utilizado para abordar este problema y cuales son las métricas que nos permiten evaluar la detección. Las soluciones que aporta el *deep learning* para resolver estos problemas las veremos en la siguiente sección.

2.4.1. Problemas de la detección

Como hemos visto antes, problemas relacionados con la visión que para las personas resultan una tarea trivial de resolver, no lo son tanto para los ordenadores. Además de los problemas típicos de la visión por computador vamos a mencionar algún problema más relacionado con la detección de objetos.

Uno de los problemas en la detección de objetos es el número variable de objetos. Al entrenar modelos de aprendizaje automático, generalmente la salida tiene un tamaño fijo. Como la cantidad de objetos en la imagen no se conoce de antemano, no conocemos la cantidad correcta de salidas. Debido a esto, se requiere algo de post-procesado, lo que agrega complejidad al modelo.

Otro gran problema son los diferentes tamaños que pueden tomar los objetos. En la tarea de clasificación, el algoritmo se centra en clasificar los objetos que cubren la mayor parte de la imagen, es decir, el objeto principal de la imagen. Por el contrario en la detección, es posible que algunos de los objetos que se deseé encontrar estén en segundo plano.

Una tercera dificultad es resolver dos problemas al mismo tiempo. En concreto, cómo combinar los dos tipos diferentes de requisitos: ubicación y clasificación en, idealmente, un único modelo.

Antes de sumergirse en como el *deep learning* se enfrenta a estos desafíos, hagamos un rápido repaso de los métodos clásicos para detectar objetos.

2.4.2. Enfoque clásico

Aunque ha habido muchos y diferentes tipos de métodos a lo largo de los años, queremos centrarnos en los dos más populares.

El primero es el marco Viola-Jones propuesto en 2001 por Paul Viola y Michael Jones [40]. El enfoque es rápido y relativamente simple, tanto que es el propio algoritmo implementado, el que permite la detección de rostros en tiempo real con poca potencia de procesamiento. Este marco combina descriptores Haar [27] y el clasificador adaboost [41] en distintas regiones propuestas por un algoritmo de selección.

Otro de los métodos tradicionales que se ha estado usando hasta hace poco es la ventana deslizante con un clasificador que predice si en esa ventana está el objeto buscado o no. Para ello se ha hecho uso de descriptores como Haar [27] o HOG [42], y de SVM como clasificador. El método de la ventana deslizante se combina con la pirámide de imágenes [43], que resuelve el problema de las distintas escalas, y con la eliminación de cuadrados múltiples (técnica conocida como *non maximum suppression* [44]).

Para ver el desarrollo y uso de la ventana deslizante, y de la pirámide de imágenes, se ha desarrollado un notebook en el que se muestra como están definidas estas técnicas y cómo aplicarlas sobre imágenes. Como clasificador se utiliza una red convolucional previamente entrenada, y además se aplica la técnica de *non maximum suppression*.

Como se puede ver en el notebook, los problemas que tienen estos métodos es que son muy lentos, además de que es tedioso fijar tamaños de ventana y de pirámide, y que debido a estas configuraciones de parámetros nos podemos dejar cosas sin detectar.

La solución que plantea el *deep learning* para intentar resolver estos problemas es entrenar el modelo de principio a fin para detectar objetos. Pero antes de pasar a explicar las técnicas de *deep learning* para detección de objetos, vamos a ver cómo se evalúa un modelo de detección.

2.4.3. Evaluación de la detección

Para evaluar el rendimiento de un detector de objetos se usa una métrica conocida como intersección sobre unión, en inglés *intersection over union* o IOU. La IOU es una medida de evaluación que se utiliza para medir la precisión de un detector de objetos en un conjunto de datos particular. A menudo vemos esta medida de evaluación utilizada en los desafíos de detección de objetos, como el popular desafío PASCAL VOC [45]. Cualquier algoritmo de

detección que proporcione un cuadro delimitador predicho como salida se puede evaluar utilizando IOU [46].

Dado un rectángulo que representa la posición correcta de un objeto (a este rectángulo se le conoce como *ground truth*) y un rectángulo que representa la predicción del modelo, la IOU se define de la manera explicada en la figura 12:

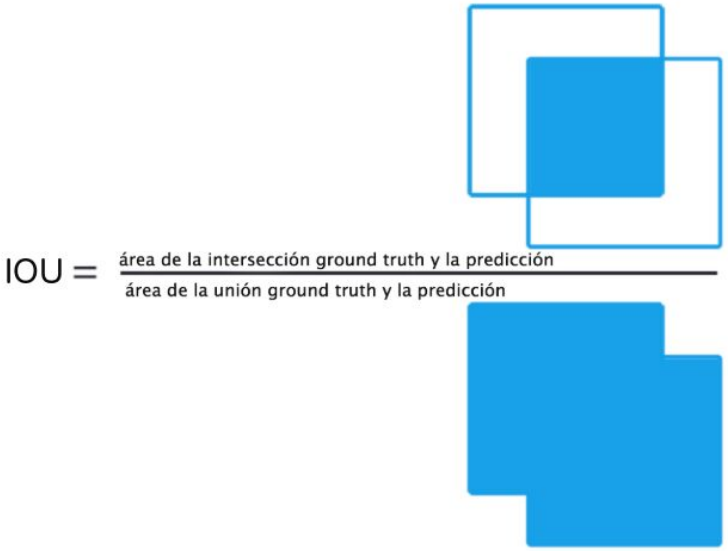


Figura 12. Definición de la métrica IOU.

La IOU se define como una fracción. En el numerador, calculamos el área de superposición entre el cuadro delimitador predicho y el cuadro delimitador del *ground truth*. El denominador es el área de unión, es decir, el área comprendida tanto por el cuadro delimitador predicho como por el recuadro delimitador del *ground truth*.

La razón por la que necesitamos está métrica es que en evaluaciones de aprendizaje automático, concretamente en la clasificación, el rendimiento de los modelos es muy sencillo de evaluar (por ejemplo, es fácil comprobar si una imagen contiene un gato o no); sin embargo, para la detección de objetos no es algo tan sencillo. En concreto, es complicado que las coordenadas (x, y) de nuestro rectángulo predicho coincidan exactamente con las coordenadas (x, y) del rectángulo que representa al objeto.

Debido a los diferentes parámetros de nuestro modelo, una coincidencia completa y total entre los recuadros es poco realista. Debido a esto, necesitamos definir una métrica de evaluación como es el caso de la IOU que nos permite decir si una detección es mala, buena o excelente. Para ello se fija un valor mínimo, normalmente 0,5; como puede verse en el ejemplo de la figura 13.

Figura 13. Ejemplo de uso de IOU, considerando que si IOU > 0,5 es un buen resultado.

Sin embargo, la IOU no tiene en cuenta la clase y se define para un solo objeto, mientras que cuando evaluamos un modelo de detección queremos hacerlo con respecto a un conjunto de test y teniendo en cuenta la clase de los objetos detectados, por lo que la métrica de IOU no se puede utilizar directamente. Para resolver estos problemas se define la métrica mAP (*mean Average Precision*) [47], que se define a continuación.

Supongamos que queremos evaluar nuestro detector de objetos en un conjunto de imágenes de test T y que en dichas imágenes pueden aparecer objetos de un conjunto de clases ζ con $\#|\zeta|=n$.

Comenzamos definiendo lo que es una detección correcta. Dado el *ground truth* de un objeto de una clase $C \in \zeta$, diremos que dicho objeto es detectado correctamente si nuestro modelo produce una predicción tal que la IOU del *ground truth* con una caja predicción es mayor que 0,5 y C es igual a la clase predicha. El umbral con el que se trabaja habitualmente es 0,5; aunque este valor puede variar.

Ahora calculamos el valor de IOU para cada rectángulo del *ground truth*. Usando este valor y nuestro umbral IOU (digamos 0,5), calculamos el número de detecciones correctas para cada clase en una imagen.

A continuación, para cada una de las imágenes, obtenemos la cantidad de objetos reales de una clase $C \in \zeta$ dada en esa imagen I; y definimos la *precision* de la clase C en una imagen I de la siguiente manera:

$$Precision_{C,I} = \frac{nº\ de\ detecciones\ correctas\ de\ la\ clase\ C\ en\ I}{nº\ de\ objetos\ de\ la\ clase\ C\ en\ I}$$

Como estamos interesados en evaluar el modelo en el conjunto de imágenes T, definimos la *Average Precision* para la clase C como:

$$Average\ Precision_C = \frac{\sum_{I \in T} Precision_{C,I}}{nº\ total\ de\ imágenes\ de\ T\ con\ objetos\ de\ la\ clase\ C}$$

Para representar el rendimiento global de nuestro modelo realizaremos la media de todas las *Average Precision* de todas las clases que tengamos. Para eso se utiliza la métrica *mean Average Precision* (mAP) definida como:

$$mAP = \frac{\sum_{C \in \zeta} Average\ Precision_C}{nº\ de\ clases}$$

En ocasiones al mAP se le añade el valor de threshold usado por la IOU y se denota por mAP@0.5.

Otros parámetros que nos pueden ayudar a evaluar la eficacia del modelo son la *precision* (no confundir con la precision de la clase `C` en una imagen `I`) que es la relación entre el número de detecciones correctamente predichas y las detecciones totales predichas; el *recall*, que es la relación entre el número de detecciones correctamente predichas y las detecciones totales; y el *F1-score* que tiene en cuenta la *precision* y el *recall*. Vamos a definir estas métricas de manera más formal, para lo cual es necesario definir los siguientes conceptos:

- *True Positive* (TP): que son los objetos detectados correctamente.
- *False Positive* (FP): son los objetos predichos por el modelo pero que no están en el *ground truth.*
- *False Negative* (FN): que son los objetos que están en el *ground truth* pero que no han sido predichos por el modelo.

Entonces podemos definir el *precision,* el *recall* y el *F1-score* de la siguiente manera:

$$Precision = \frac{TP}{FP+TP}$$

$$Recall = \frac{TP}{FN+TP}$$

$$F1-score = \frac{2*TP}{2*TP+FP+FN}$$

Una vez presentado el contexto y las nociones mínimas necesarias para entender la detección de objetos, pasamos a explicar las técnicas de *deep learning* para resolver este tipo de problemas.

3. Deep Learning para la detección de objetos

Hace poco tiempo que las técnicas de *deep learning* se han empezado a utilizar en la detección de objetos [48,49]. Estas técnicas quedan divididas en dos categorías:

1. *Los algoritmos de dos fases*, cuyo primer paso consiste en generar un grupo de posibles rectángulos delimitadores o propuestas de regiones "interesantes" y que son las que se clasifican usando redes neuronales convolucionales en el segundo paso. Por ejemplo, son algoritmos de dos fases la R-CNN [48], Fast R-CNN [50] y Faster R-CNN [51].
2. *Los algoritmos de una fase* dividen la imagen en regiones, esas regiones pasan a una red convolucional, y por último dichas regiones se modifican y agrupan basándose en la predicción obtenida. Por ejemplo, algunos algoritmos de una fase son SSD [52] y YOLO [49].

En general, los primeros algoritmos son más precisos pero más lentos (debido a las dos fases) por lo que dan problemas para el procesado de imágenes en tiempo real. Por otro lado, los segundos son más rápidos aunque son algo menos precisos debido a que todo se hace a la vez.

Recientemente el algoritmo YOLO, un algoritmo de una fase, ha alcanzado una precisión comparable a los de dos fases, pero manteniendo el procesado en tiempo real, ver tabla 1. Es por esto que hemos elegido este algoritmo para nuestro trabajo.

Detection Framework	**mAP**	**FPS**
Faster RCNN - VGG16	73,2	7
Faster RCNN - ResNet	76,4	5
YOLO v1	63,4	45
SSD 500	76,8	19
YOLO (416×416 image size)	76,8	67
YOLO (480×480 image size)	77,8	59

Tabla 1. Tabla comparativa de precisión y tiempo de algoritmos de detección en el dataset Pascal Voc [45].

El resto de esta sección se organiza del siguiente modo. Primero, se va a explicar cómo se puede abordar el problema de la localización. A continuación, se introduce cómo las ideas aplicadas para la localización de objetos se extienden al problema de la detección de objetos en la red YOLO. Por último, veremos cómo trabajar con YOLO para detectar objetos usando una versión pre-entrenada de la red. Para más información del resto de algoritmos de *deep learning* para la detección de objetos consultar los anexos.

3.1. Localización de objetos

Como hemos explicado anteriormente, la localización de objetos consiste en la capacidad de un modelo para identificar donde está el objeto principal dentro de la imagen colocando un

rectángulo delimitador a su alrededor. La posición dentro de una imagen se define mediante el par (b_x, b_y), que representan el punto medio del objeto, y los valores b_w y b_h que representan el ancho y el alto del rectángulo respectivamente.

Además de localizar dónde se encuentra el objeto, también se puede querer clasificar dicho objeto en una de n posibles categorías. Para ello usamos el *one-hot encoding* [53], es decir, si un objeto pertenece a la categoría i esto lo representamos mediante un vector cuya componente i-ésima es 1 y el resto 0.

Entonces dada una imagen como entrada, queremos construir un modelo que produzca como resultado un vector de características con la siguiente forma:

$$y = \begin{bmatrix} p \\ b_x \\ b_y \\ b_w \\ b_h \\ c_1 \\ . \\ . \\ . \\ c_n \end{bmatrix}$$

Donde p indica el nivel de confianza del modelo sobre si hay un objeto en la imagen o no (si $p = 1$ indica que el modelo está completamente seguro que la imagen contiene al objeto y si $p = 0$ está completamente seguro de que no lo contiene). Como hemos dicho los valores $b_x, b_y, b_w \; y \; b_h$ forman el rectángulo delimitador alrededor del objeto, y los valores $c_1, c_2, \ldots \; y \; c_n$, representan la categoría del objeto usando *one-hot encoding*. Por ejemplo, si queremos localizar y clasificar un coche que aparece en el centro de una imagen (por lo que los valores de b_x y b_y son 0,5; 0,5), el ancho del coche es el 40% de la imagen (por lo que el valor de b_w es 0,4) y su alto el 20% (el valor de b_h es 0,2) y donde las posibles categorías son moto, coche y bici, el vector de características resultante tendrá la siguiente forma:

$$y = \begin{bmatrix} 1 \\ 0.5 \\ 0.5 \\ 0.4 \\ 0.2 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

En caso de que el modelo no detecte ningún objeto, la salida sería:

$$y = \begin{bmatrix} 0 \\ ? \\ ? \\ ? \\ ? \\ ? \\ ? \\ ? \end{bmatrix}$$

Donde solo interesará el valor de p y el resto de valores no, lo que se denota en el vector anterior con interrogantes.

Una vez explicado cómo localizar y clasificar objetos en un imagen podemos pasar a explicar como funciona el algoritmo YOLO.

3.2. YOLO

YOLO [49] toma un enfoque completamente diferente a los métodos de detección tradicionales. No es un clasificador tradicional que se reutiliza para ser un detector de objetos. *YOLO* mira la imagen solo una vez (de ahí su nombre: *You Only Look Once*) pero de una manera inteligente. Aunque existen varias versiones de YOLO, las ideas principales son las mismas para todas ellas.

El algoritmo YOLO funciona de la siguiente manera, comienza colocando una cuadrícula (o *grid*) encima de la imagen de entrada. Este grid divide la imagen de entrada en una cuadrícula $m_1 \times m_2$, como podemos ver en la figura 14.

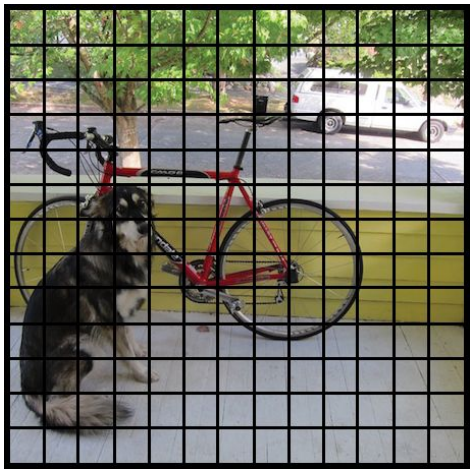


Figura 14. División de la imagen por el grid.

Para cada una de estas celdas del grid, se ejecuta el algoritmo de localización que hemos visto en el apartado anterior y se obtiene un vector con la siguiente forma:

$$y = \begin{bmatrix} p \\ b_x \\ b_y \\ b_w \\ b_h \\ c_1 \\ . \\ . \\ . \\ c_n \end{bmatrix}$$

Como un objeto puede ocupar varias celdas, la celda responsable de detectar ese objeto será aquella en la que se encuentra el centro del objeto. Notar que los valores de b_x, b_y, b_w y b_h se dan en función del tamaño de la celda, por lo que los valores de b_x y b_y siempre estarán entre 0 y 1, pero los valores de b_w y b_h pueden ser superiores a 1 (en caso de que el objeto no entre en la celda).

Por lo tanto lo que queremos construir con YOLO es un modelo que produzca como salida un tensor (una generalización de la noción de matriz a dimensiones superiores) de tamaño $m_1 \times m_2 \times (5+n)$. Para cada una de las $m_1 \times m_2$ celdas del grid se obtiene un vector con la probabilidad del objeto; su posición en términos de b_x, b_y, b_w y b_h; y la categoría codificada usando *one-hot encoding*, donde n es el número de categorías posibles.

Uno de los problemas de la detección de objetos usando la aproximación que acabamos de explicar es el hecho de que cada celda del grid solo puede detectar un objeto. Entonces si se tienen varios objetos en la misma celda, la técnica que hemos utilizado hasta ahora no permite detectarlos. Las cajas de anclaje [54] nos ayudan a resolver este problema.

La idea aquí es predefinir rectángulos (llamados cajas de anclaje) de diferentes formas para cada objeto y asociar predicciones a cada una de ellas, ver figura 15.

Figura 15. Ejemplo de cajas anclaje en una imagen. Como se puede apreciar en la imagen, las cajas de anclaje tienen distintos tamaños y orientaciones.

Nuestro vector de salida ahora tendrá $5+n$ dimensiones más por cada una de las cajas de anclaje que predefinamos. Es decir, ahora en lugar de predecir para cada celda un vector de

tamaño $5+n$, se predice un vector de la siguiente forma (el siguiente ejemplo corresponde a dos cajas de anclaje):

$$y = \begin{bmatrix} p_1 \\ b_{x_1} \\ b_{y_1} \\ b_{w_1} \\ b_{h_1} \\ c_1 \\ . \\ . \\ c_n \\ p_2 \\ b_{x_2} \\ b_{y_2} \\ b_{w_2} \\ b_{h_2} \\ c_1 \\ . \\ . \\ c_n \end{bmatrix}$$

Es decir, el modelo predice un tensor de tamaño $m_1 \times m_2 \times (k \times (5+n))$ donde k es el número de cajas de anclaje.

Utilizando estas ideas podríamos detectar un objeto varias veces, ya que es posible que muchas celdas detecten los objetos. Para evitar eso, y como cada predicción tiene, como hemos dicho antes, un valor p que identifica la probabilidad de predicción, se descartaran todas aquellas cuya probabilidad sea inferior a un *threshold* dado. Aunque el paso anterior elimina muchos rectángulos, es posible que todavía queden muchos y que se superpongan entre ellos. Para solucionar este problema miramos los rectángulos que más se superponen con el de "mayor probabilidad" y se eliminan aquellos que tienen un alto IOU con respecto al rectángulo de mayor probabilidad. Finalmente, los rectángulos restantes son la detección correcta, a esta técnica se la conoce como *non-maximum supression [44]* y obtenemos el resultado final, ver figura 16.

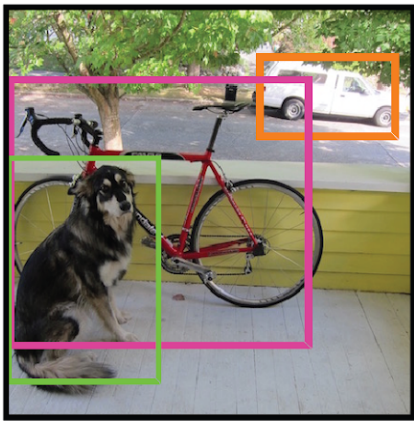


Figura 16. Detecciones de alta puntuación.

Hasta ahora nos hemos centrado en la salida del modelo YOLO, pero no hemos hablado de la arquitectura que utiliza. En concreto, el modelo YOLO está basado en una red convolucional inspirada en GoogleNet para la clasificación de imágenes. Dicha red fue entrenada inicialmente para resolver el problema de clasificación de ImageNet y luego fue adaptada para abordar problemas de detección, ver figura 17.

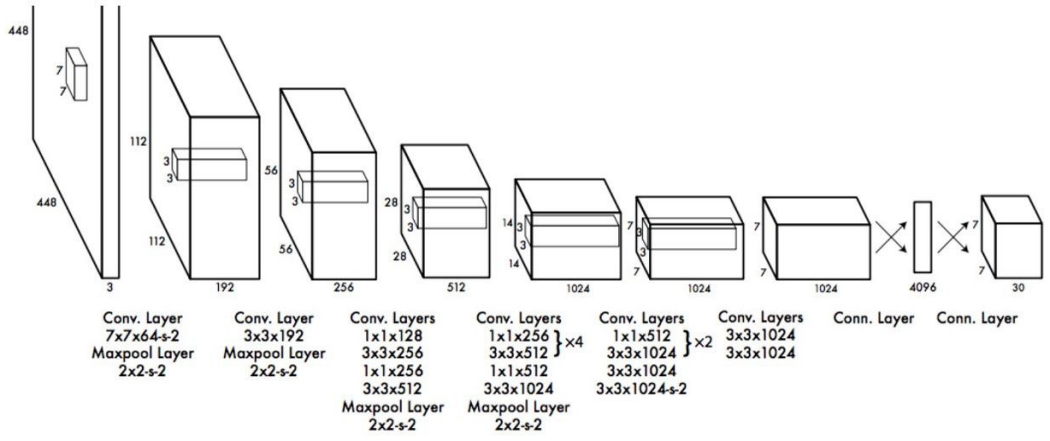


Figura 17. Arquitectura de YOLO.

En esta sección hemos explicado brevemente las ideas fundamentales de la red YOLO. Sin embargo, la red YOLO introduce muchas cuestiones técnicas, y de bajo nivel, para mejorar el proceso de detección. Dichas cuestiones quedan fuera del alcance de este trabajo y pueden consultarse en la documentación de YOLO [49].

3.3. Usando YOLO

Para terminar esta sección, se ha desarrollado un notebook donde se usa YOLO. Al descargar YOLO, este modelo está pre-entrenado para el dataset de CoCo [55] (hablaremos más adelante y de forma detallada sobre este dataset), por lo que puede ser utilizado directamente para hacer predicciones sobre unas categorías determinadas.

En dicho notebook se encuentran las instrucciones necesarias para descargar YOLO y para usarlo sobre las imágenes que se usaron en el notebook en el que se explicaba el uso de la ventana deslizante. Aunque el proceso para aplicar YOLO viene descrito en el notebook, se termina esta sección mostrando y comparando los resultados obtenidos por la ventana deslizante y YOLO, ver tabla 2.

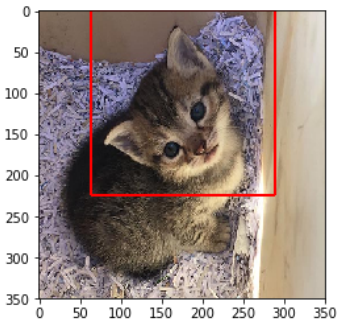
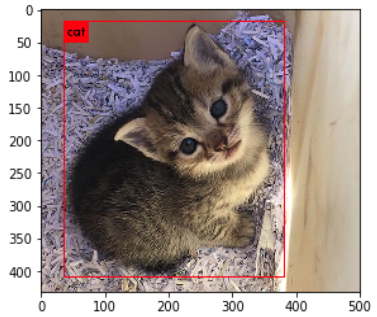
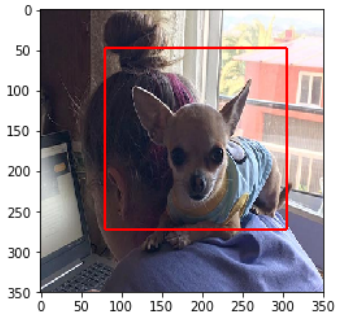
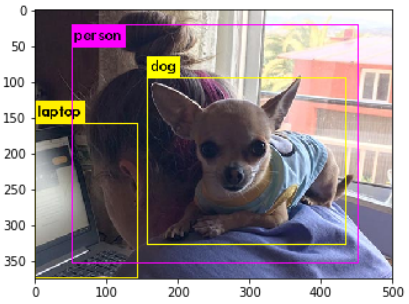
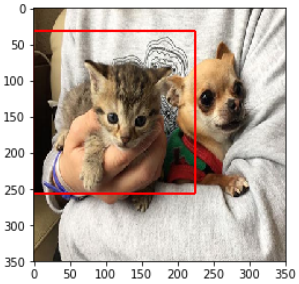
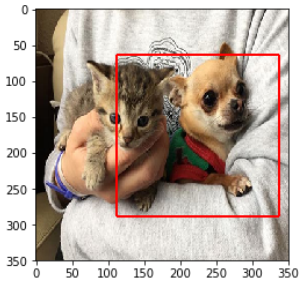
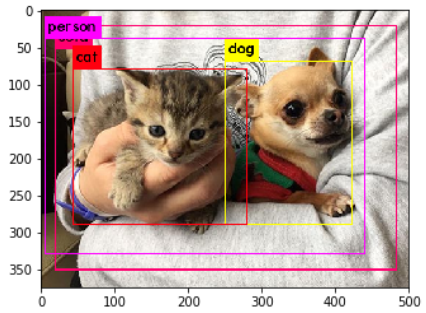
Ventana Deslizante	Yolo
	
Categoría: gato persa Tiempo = 169 s	Tiempo = 14 s
	
Categoría: chihuahua Tiempo = 463 s	Tiempo = 13,96 s
 	
Categoría: lince Categoría: chihuahua Tiempo = 115 s	Tiempo = 14,04 s

Tabla 2. Tabla comparativa de tiempos y resultados de la ventana deslizante y YOLO.

En el caso de la ventana deslizante las categorías pueden ser más precisas, ya que el modelo es capaz de predecir más categorías, pero la detección es más precisa si utilizamos YOLO; además de ser más rápido, como se puede observar en la tabla anterior (ya que YOLO es aproximadamente 10 veces más rápido que la ventana deslizante, además de que detecta múltiples objetos en la misma imagen). Esto muestra las ventajas de utilizar YOLO en lugar de la aproximación tradicional.

En la siguiente sección veremos cómo entrenar YOLO con nuestros propios datasets.

4. Entrenando YOLO

Si recordamos el problema principal de este proyecto es realizar un asistente que vaya guiando a usuarios no expertos en los pasos necesarios para realizar la detección automática de objetos en imágenes o vídeos. Este tipo de problemas se abordan siguiendo el flujo de trabajo presentado en la figura 18.

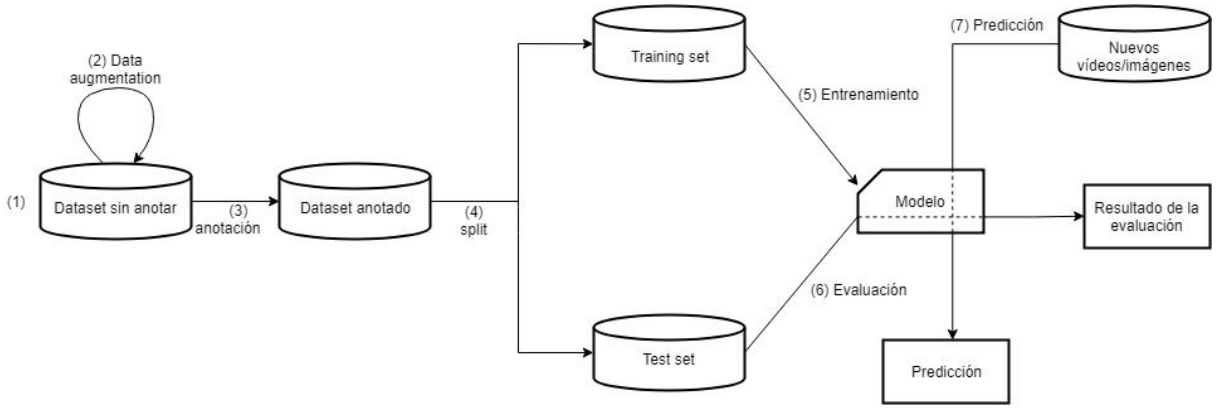


Figura 18. Diagrama del flujo de trabajo para construir y utilizar un modelo de detección.

En esta sección, explicamos en detalle cómo cada uno de los pasos de este flujo de trabajo se traslada a nuestro proyecto. En concreto, para cada punto se explica en qué consiste dicho paso, las alternativas para abordarlo, las particularidades de dicho punto por trabajar con la red YOLO (que se puede descargar desde la página web de sus autores en https://pjreddie.com/darknet/yolo/), y las soluciones que nosotros aportamos.

El proceso para crear un modelo de detección en vídeos consiste en crear un modelo de detección en imágenes y luego aplicar dicho modelo para detectar objetos en cada uno de los *frames* del vídeo. Por lo tanto a partir de ahora nos centramos en explicar cómo construir un modelo de detección de objetos en imágenes.

4.1. Recolección del dataset

El primer paso para construir el modelo de detección será reunir un conjunto de imágenes iniciales. Este dataset inicial debe contener imágenes que contengan imágenes positivas (es decir, un conjunto de imágenes que contienen los objetos a detectar) e imágenes negativas (es decir, un conjunto de imágenes que no contienen los objetos a detectar). El número de imágenes debe ser aproximadamente uniforme, es decir, el dataset debe contener el mismo número (o al menos un número parecido) de imágenes para cada una de las categorías de

los objetos que se quiera detectar. Notar que este paso es independiente del modelo de *deep learning* que vayamos a utilizar posteriormente.

Aunque este paso parece obvio, es uno de los que más complicaciones tiene y más tiempo ocupa, ya que es necesario buscar imágenes positivas y negativas, y recoger la cantidad necesaria de imágenes. Además, como hemos contado en secciones anteriores, este proceso tiene una gran cantidad de problemas debido al número variable de objetos y a los distintos tamaños de los objetos, por lo que es necesario construir un dataset que abarque el mayor número de situaciones posibles. Además, hay que sumarle los problemas de conseguir imágenes que tengan relación con el problema que se está tratando.

Podemos recolectar las imágenes desde muchas fuentes, a continuación se enumeran las distintas alternativas que han sido probadas hasta encontrar la mejor solución.

1. *Búsqueda en internet de bancos de imágenes libres*. Esta opción consiste en localizar datasets de imágenes en Internet que contengan y no contengan los objetos a detectar.
2. *Descargar individualmente desde Internet las imágenes que contengan los objetos*. Utilizando un motor de búsqueda, como Google imágenes, se puede descargar una a una imágenes que contengan y no contengan los objetos a detectar. Las imágenes a utilizar en esta opción deberán tener una licencia del estilo *Creative Commons* o Documentación libre de GNU para no infringir derechos de autor.
3. *Bancos de imágenes privados*. En algunos casos, por ejemplo en el caso de imágenes médicas, son los usuarios finales los que deben proporcionar las imágenes ya que imágenes de ese tipo no son de dominio público.
4. *Utilizar dispositivos dedicados*. En este caso nosotros mismos podemos obtener las imágenes que sean necesarias, con y sin el objeto. Como por ejemplo en las líneas de producción, puede ser útil emplear dispositivos dedicados a la captura de imágenes.
5. *Descargar vídeos que contengan y no contengan los objetos a detectar.* En esta opción se descargan vídeos de Internet y se utilizan los *frames* que no contienen los objetos para la obtención de las imágenes negativas; y para el caso de las imágenes positivas se ven los vídeos y se guardan aquellos *frames* que sí contengan el objeto.

Las dos primeras opciones son en general desechadas. La primera debido a la falta de conjuntos de imágenes específicos que posean los objetos buscados, y la segunda debido a la gran pérdida de tiempo, a la poca eficacia que esto supone y a la existencia de otras posibilidades. La tercera opción se utiliza cuando el cliente puede proporcionar las imágenes y la cuarta cuando se pueden instalar dispositivos. Estas dos opciones tienen la ventaja de que las imágenes son parecidas a las que luego el modelo se va a encontrar en producción por lo que es recomendable usarla en esas situaciones. En caso de que estas alternativas no sean factibles, la quinta opción es la más rápida y óptima.

Con respecto al número de imágenes que debe contener el dataset, en nuestra experiencia con estos proyectos se suelen coger 3000 imágenes por cada categoría de objeto a detectar

(es decir si hay 10 categorías, el conjunto de imágenes positivas sería aproximadamente de 30000); y en caso de la muestra negativa, se suele tener el mismo número de negativas que el total de las positivas.

4.2. Data augmentation

En caso de que el número de imágenes no sea el suficiente, podemos usar la técnica conocida como aumento de datos o *data augmentation* [56]. El aumento de datos crea artificialmente imágenes de entrenamiento a través de diferentes transformaciones, como rotaciones, distorsiones, etc, con el fin de conseguir un tamaño de muestra mayor.

Las técnicas más usadas para el aumento del dataset son las siguientes:

- *Voltear*: podemos darle la vuelta a las imágenes horizontalmente y verticalmente, como se puede ver en la figura 19.

Figura 19. De izquierda a derecha, la imagen original, la imagen volteada horizontalmente, y la imagen volteada verticalmente.

- *Rotar*: giramos la imagen un ángulo dado, como en la figura 20. Hay que tener en cuenta que en caso de la imagen sea cuadrada no sufrirá ninguna modificación en cuanto a sus dimensiones, en caso de que sea rectangular sus dimensiones después de la rotación no serán las mismas. Para evitar hacer relleno de las imágenes, las imágenes solo se suelen girar 90, 180 y 270 grados.

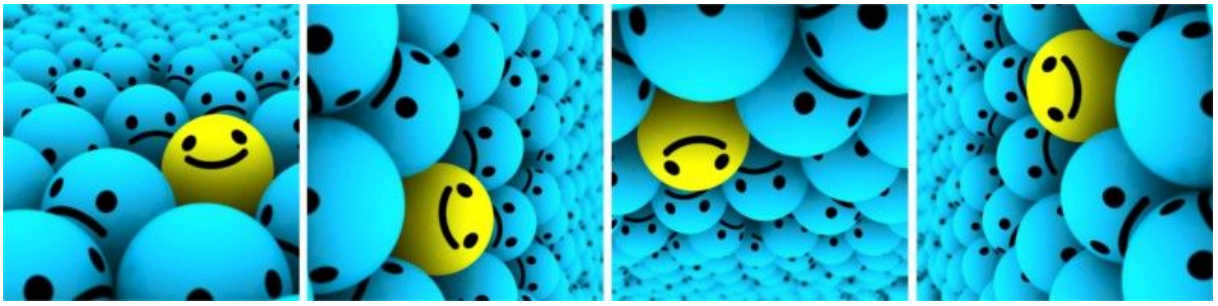


Figura 20. Rotación de la imagen con un ángulo de 90^o en el sentido de las agujas del reloj.

- *Escalar*: podemos reescalar la imagen tanto hacia fuera (ampliando) como hacia dentro (reduciendo), ver figura 21.

Figura 21. De izquierda a derecha, la imagen original, la imagen ampliada un 10% y ampliada un 20%.

- *Traslación*: la traslación implica mover la imagen a lo largo del eje X o Y (o en ambos). Para la figura 22, suponemos que la imagen tiene un fondo negro que se extiende más allá de su límite, y se traslada apropiadamente.

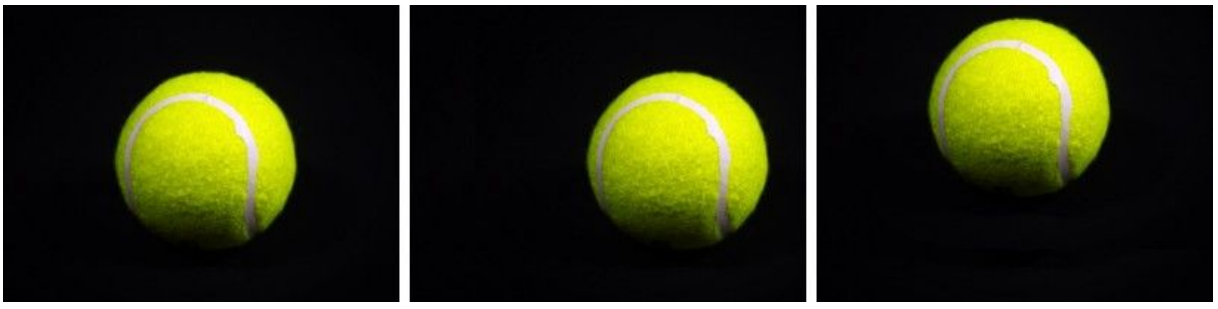


Figura 22. De izquierda a derecha, la imagen original, la imagen trasladada a la derecha y trasladada hacia arriba.

- *Filtros*: a parte de las técnicas vistas antes, también se le pueden aplicar a la imagen filtros. En el caso de la figura 23 se le está aplicando a la imagen el ruido de sal y pimienta, y el ruido Gaussiano.

Figura 23. De izquierda a derecha, la imagen original, la imagen con un ruido Gaussiano y con ruido de sal y pimienta.

Este paso es independiente del modelo de *deep learning* que se utilice. Si el dataset no es lo suficientemente grande para entrenar el algoritmo usaremos estas técnicas para ampliarlo. Otra opción que se puede utilizar para ampliar el dataset consiste en emplear un programa

llamado *ffmpeg* [57] que permite colocar el objeto recortado en imágenes negativas, además de rotarlo, con el fin de obtener más muestras positivas.

4.3. Anotación del dataset

Una vez que tenemos las imágenes es necesario que las anotemos, es decir, que proporcionemos de manera manual a cada imagen una etiqueta. Estas etiquetas indican la posición de cada objeto dentro de la imagen y la categoría a la que pertenece cada uno de dichos objetos. El problema que tiene esta etapa es que es muy costosa (notar que si tenemos 30000 imágenes, tendremos que anotar todas ellas de manera manual). Además, en algunos casos (por ejemplo en imágenes médicas) requieren del conocimiento de un experto para anotarlas correctamente.

Cada modelo de detección utiliza un formato de anotación concreto, en nuestro caso, la red YOLO requiere de un archivo `.txt` para cada una de las imágenes con una línea para cada objeto a detectar, con la siguiente forma:

```
<object-class> <x> <y> <width> <height>
```

Donde `<object-class>` es la clase del objeto, `<x>` e `<y>` son las coordenadas del centro del objeto y `<width>` y `<height>` son el ancho y el alto del rectángulo que contiene al objeto. Estos archivos no se escriben de manera manual, sino que se anotan utilizando herramientas de anotación, por ejemplo *LabelImg* [58] o *YOLO mark* [59], y luego se ejecutan scripts para generar todos los archivos necesarios.

Desafortunadamente, no existe un formato de anotación estándar, y de hecho varía entre los *frameworks* de detección de objetos, y también entre las herramientas de anotación. Por lo tanto, en caso de que la herramienta de anotación no genere los ficheros en el formato correcto, será necesario un paso de conversión utilizando *scripts* como los proporcionados en [60].

Como hemos dicho antes este es un proceso largo, ya que supone recorrer todas las imágenes positivas indicando donde se encuentra cada uno de los objetos de la imagen.

4.4. Dataset split

Como en cualquier modelo de aprendizaje automático, es muy importante descomponer el conjunto de datos en dos partes:

- El conjunto de entrenamiento: conjunto de imágenes que utilizaremos para el entrenamiento del algoritmo.
- En conjunto de evaluación: conjunto de imágenes que utilizaremos para la evaluación del algoritmo.

La idea es dividir nuestro conjunto de imágenes, en uno o varios conjuntos de entrenamiento y otros conjuntos de test como por ejemplo pueden ser los de la figura 24. Es

decir, no le vamos a pasar todas nuestras imágenes al algoritmo durante el entrenamiento, sino que vamos a retener una parte de las imágenes para realizar una valoración del rendimiento del modelo y poder observar si funciona como debe.

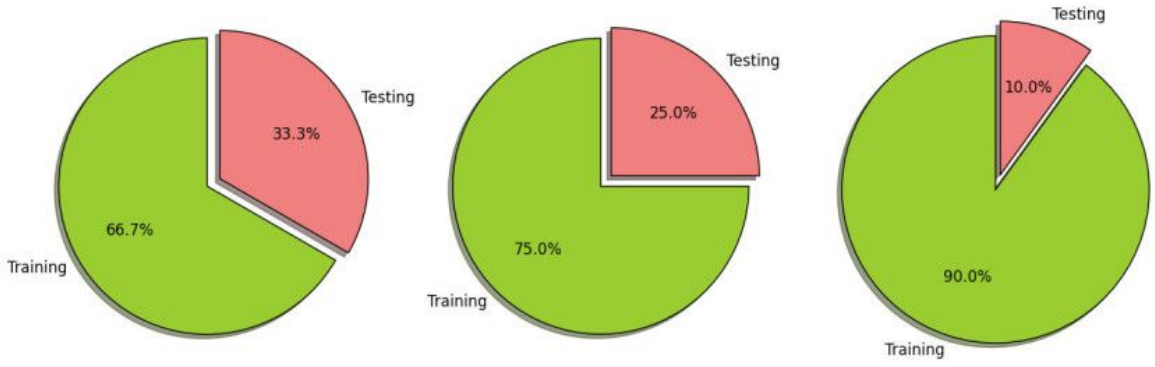


Figura 24. Alternativas de las posibles divisiones de nuestro conjunto de datos.

En algunos casos es necesario realizar modificaciones en los parámetros de los algoritmos de clasificación, con el fin de encontrar los parámetros adecuados para su mejor funcionamiento. Para este caso el conjunto de entrenamiento vuelve a sufrir otra división donde cogeremos entre un 10-20% de las imágenes del conjunto de entrenamiento, este nuevo conjunto es conocido como *conjunto de validación*. Con esta nueva división lo que buscamos es evitar que los mismos datos que usamos para entrenar sean los mismos que utilizamos para evaluar. De esta forma, vamos a poder analizar con más precisión como el modelo se va comportando a medida que lo vamos entrenando y poder detectar el punto crítico en el que el modelo deja de generalizar y comienza a *sobre ajustarse* a los datos de entrenamiento.

En YOLO este proceso es manual ya que tenemos que ser nosotros mismos los que dividamos el dataset y almacenemos las imágenes de entrenamiento en una carpeta y las de test en otra distinta. El proceso es delicado ya que en caso de que no se organicen de manera correcta las carpetas para el proceso de entrenamiento, este falla.

4.5. Entrenamiento

Después de procesar las imágenes (es decir, organizarlas en sus carpetas correspondientes y anotarlas) se entrena un modelo capaz de detectar y clasificar los objetos dentro de una imagen. Aquí es donde realmente comenzamos a utilizar las técnicas de *deep learning*.

Entrenar una red neuronal artificial es un proceso en el que se desea encontrar una configuración óptima de pesos, de tal manera que la red pueda extraer información útil (o patrones) de los datos que le pasamos por entrada, para luego poder generalizar. Los pesos son coeficientes que se van adaptando dentro de la red a medida que esta va aprendiendo a partir de los ejemplos de entrenamiento. Ellos son la medida de la fuerza de una conexión

de entrada. Estas fuerzas pueden ser modificadas en respuesta de los ejemplos de entrenamiento de acuerdo a la topología específica o debido a las reglas de entrenamiento.

En el caso de YOLO existen dos opciones para llevar a cabo el entrenamiento, YOLO se puede entrenar desde cero o se pueden utilizar unos pesos previamente entrenados (que provienen del modelo darknet53 [61]) para las capas convolucionales. En ambos casos es necesario pasarle como entrada las imágenes anotadas. Es preferible usar la segunda opción de entrenamiento sobre la primera ya que se parte de una red que ya ha sido entrenada y que no le costará demasiado encontrar la configuración óptima de pesos, ya que si se parte desde cero el entrenamiento puede suponer mucho tiempo.

Para configurar el entrenamiento de la red YOLO se utiliza como base el fichero `yolo-obj.cfg` que contiene la estructura de la red YOLO, y además una serie de parámetros a modificar. Para realizar dichas modificaciones nos basamos en lo explicado en la documentación de YOLO [62] y en [63].

- *batch:* Este parámetro es lo que se conoce como *batches* o *batch size* y se utiliza para que en el entrenamiento, las imágenes se muestren en conjuntos (conocidos como *batches*) a la red y no de una en una (ya que esto supondría realizar muchos cambios en los pesos de la red) ni todas a la vez (algo que no es factible debido a límites de memoria). Para realizar el entrenamiento el parámetro *batch* debe tener como valor 64, ya que por defecto viene a 1 que es el valor que permite realizar el test.
- *subdivision:* Este parámetro se corresponde con el número de grupos en el que dividiremos los *batches* para acelerar el proceso de entrenamiento y para mejorar la generalización. Igual que en el caso de *batch*, es necesario modificar el valor de *subdivision*, en este caso por 8.
- *classes*: este parámetro se corresponde con el número de clases que se quiere clasificar.
- *filters*: para las capas convolucionales cambiamos el número de filtros, estos filtros dependen del número de clases que vayamos a detectar y siguen la siguiente relación: *filters = (classes +5)*3.*

Además de la modificación de estos parámetros, dentro de la carpeta `darknet\data\` hay que crear una serie de ficheros:

- *train.txt*: este fichero contendrá las rutas de las imágenes que se van a utilizar para el entrenamiento. La ruta de cada imagen irá en una línea.
- *test.txt*: este es similar al anterior, pero para las imágenes que se utilizarán en el test.
- *Fichero con extensión .names*: en este fichero tenemos que poner los nombres de los objetos que queremos que YOLO detecte, cada uno en una línea. Por ejemplo en la figura 25 se muestra el fichero para el dataset CoCo donde hay 12 clases.

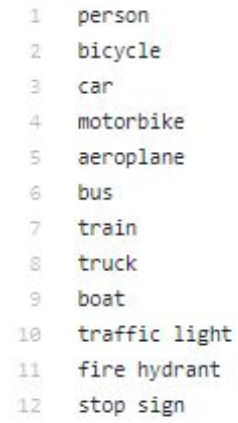


Figura 25. Fichero coco.names.

- *Fichero con extensión .data*: este fichero contendrá una serie de parámetros y rutas que hay que modificar: como el número de clases, el fichero con la lista de imágenes para entrenar, un fichero que contendrá el listado de imágenes que se usarán para realizar el test, y por último, el fichero comentado en el punto anterior con la lista de las clases a clasificar. La siguiente imagen muestra un ejemplo de dicho fichero.

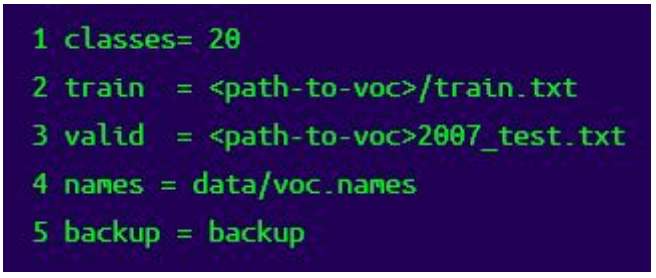


Figura 26. Fichero obj.data.

Ahora ya podemos entrenar YOLO usando la siguiente instrucción:

```
./darknet detector train cfg/voc.data cfg/yolo-obj.cfg darknet53.conv.74
```

Expliquemos un poco la instrucción anterior, `./darknet detector train` sirve para lanzar el entrenamiento; `cfg/voc.data` es el fichero en el que se encuentran las rutas de las carpetas donde están los datos, además de parámetros como el número de clases; `cfg/yolo-obj.cfg` es el archivo de configuración donde se encuentra la arquitectura de la red YOLO; y `darknet53.conv.74` son los pesos iniciales.

El proceso anterior puede finalizar de dos maneras: o bien el usuario lo detiene explícitamente o se llega a un número máximo de iteraciones. En el segundo caso el número máximo de iteraciones es un parámetro del archivo `yolo-obj.cfg` donde se indica el número de iteraciones que queremos que se realicen. Durante el proceso de entrenamiento se verán varios indicadores de error y se recomienda parar el entrenamiento cuando el valor de AVG IOU no siga disminuyendo en varias iteraciones.

Una vez terminado el proceso de entrenamiento obtendremos varios archivos .weights con los pesos (cada cierto número de iteraciones se guarda un archivo .weights) que van a permitir a la red detectar y clasificar los objetos que nos interesen. De estos pesos deberemos elegir el mejor de ellos. Con dichos archivos, YOLO ya podrá detectar y clasificar las clases que se le han indicado previamente. Para saber si la red ha sido entrenada correctamente hay que pasar a evaluarla.

4.6. Evaluación

En esta etapa ponemos a prueba la información o conocimiento que la red ha obtenido del entrenamiento en el paso anterior. Evaluamos si la red es precisa en sus predicciones y si no estamos conformes con su rendimiento, podemos volver a la etapa anterior y continuar entrenando la red cambiando algunos de los parámetros hasta lograr un rendimiento y eficacia aceptable.

El proceso de evaluación consiste en pasar las imágenes de test por la red (es muy importante que no mezclemos los conjuntos), para obtener los resultados de detección obtenidos por la red y compararlos con los de la realidad. La finalidad de este proceso nos permitirá extraer información acerca del comportamiento del modelo y datos relacionados con el rendimiento (como puede ser la tasa de aciertos o de fallos) que ayudarán a decidir si el modelo es adecuado o no para nuestro problema.

En el caso de YOLO, como hemos dicho anteriormente, durante la etapa de entrenamiento se obtienen varios ficheros .weights y tenemos que elegir el que mejor funcione. Para seleccionar uno de estos ficheros nos fijamos en el que obtenga la IoU (intersección de la unión comentada en el apartado 2.4.3) y mAP (es la media de la precisión comentada en el apartado 2.4.3) más altas. El último archivo .weigth que se ha obtenido no tiene porque ser el que mejor resultado nos de, ya que se puede producir un sobreentrenamiento, es decir, la red es capaz de detectar los objetos en las imágenes de entrenamiento pero no detecta nada en ninguna otra imagen. Para realizar la evaluación usamos el siguiente comando:

```
./darknet detector map cfg/voc.data cfg/yolo-obj.cfg yolov3.weights
```

Para ejecutar la evaluación usamos `./darknet detector map`, el fichero `voc.data` nos indica las rutas de las carpetas donde están las imágenes de test, `cfg/yolo-obj.cfg` es el archivo de configuración donde se encuentra la arquitectura de la red YOLO, y `yolov3.weights` son los pesos obtenidos en el entrenamiento.

Al ejecutar esta instrucción se nos muestra por cada una de las clases, la probabilidad de que aparezca un objeto de esa clase en una imagen del conjunto de test. Y como resultado final nos muestra el valor mAP del modelo que hemos creado indicándonos si es un buen modelo o no, ver figura 27.

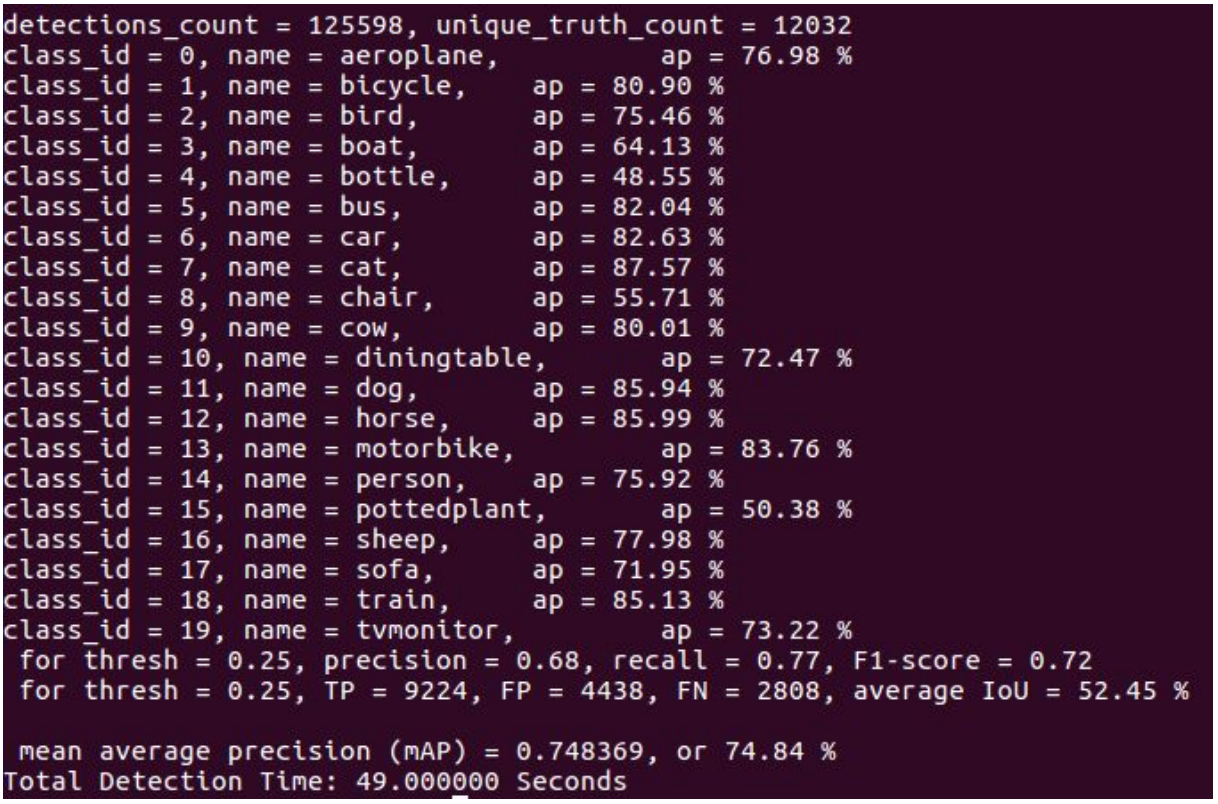


Figura 27. Resultado de la evaluación de YOLO.

En nuestro contexto consideramos que un modelo es aceptable y está listo para producción cuando se alcanza un valor de mAP del 70%. Esto puede depender del proyecto, ya que algunos de ellos, como los de análisis de imágenes médicas, requieren un mayor nivel de precisión.

4.7. Predicción

Una vez seleccionado el archivo con los mejores pesos, el siguiente paso será poner en producción el modelo y empezar a utilizarlo en imágenes que no han sido utilizadas ni para entrenar ni para realizar los tests.

Hay distintas maneras de realizar dicha predicción como se muestra a continuación. Para todos los ejemplos que mostraremos en el resto de la sección utilizaremos un modelo entrenado con el dataset de CoCo.

- Para una única imagen se usa el siguiente comando:

```
./darknet detect cfg/yolo-obj.cfg yolov3.weights data/gala.jpeg
```

Para que prediga usamos `./darknet detect`; `cfg/yolo-obj.cfg` es el archivo de configuración donde se encuentra la arquitectura de la red YOLO; `yolov3.weights` son los pesos obtenido del entrenamiento; y `data/gala.jpeg` es una imagen en la que vamos a detectar y clasificar.

YOLO muestra por consola los objetos que ha detectado, su confianza y cuánto tiempo le ha llevado encontrarlos, como podemos ver en la figura 28.

```
Loading weights from yolov3.weights...Done!
data/gala.jpeg: Predicted in 13.964199 seconds.
laptop: 100%
dog: 99%
person: 74%
```

Figura 28. Resultado de una detección.

Por defecto, YOLO no nos muestra las detecciones pero sí que las guarda en un fichero llamado `predictions.png` y desde aquí podemos ver los objetos detectados como por ejemplo en la figura 29.

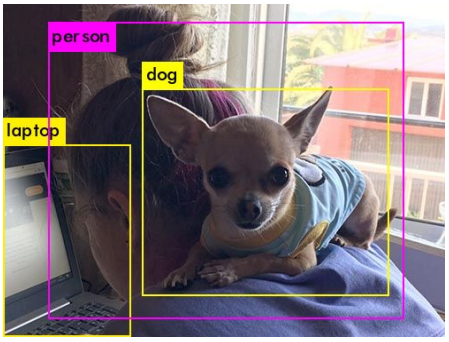


Figura 29. Imagen predictions.png.

- YOLO también puede procesar múltiples imágenes (lo que evita cargar la red cada vez que se vaya a hacer una predicción). Para ello, en lugar de pasarle una imagen al comando anterior, no se le indica nada por lo que probará con varias imágenes que le tenemos que ir pasando mediante su ruta a medida que nos las vaya pidiendo. El comando a ejecutar sería el siguiente:

```
./darknet detect cfg/yolo-obj.cfg yolov3.weights
```

Para que prediga usamos `./darknet detect`; `cfg/yolo-obj.cfg` es el archivo de configuración donde se encuentra la arquitectura de la red YOLO; `yolov3.weights` son los pesos obtenido del entrenamiento; y en este caso no ponemos la imagen sobre la que queremos que prediga si no que YOLO nos va pidiendo imágenes por la consola y así podemos predecir sobre un conjunto de imágenes. Ver ejemplo en figura 30.

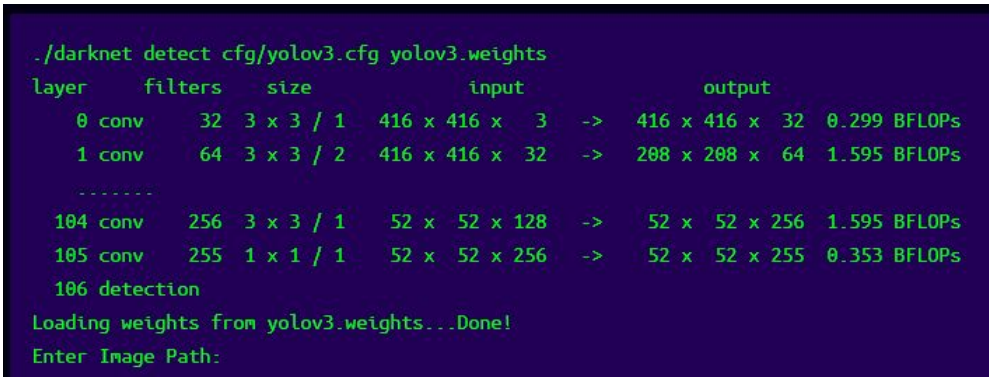


Figura 30. Ejecución de YOLO para varias imágenes.

Uno de los inconvenientes que puede tener esta opción es que como hemos dicho antes, YOLO por defecto no nos muestra las imágenes sino que las guarda, pero todas bajo el mismo nombre. Entonces no podremos ver todas las fotos que vayamos pasándole a YOLO al menos que renombremos el fichero `predictions.png` cada vez que realicemos una predicción.

- Por último YOLO también puede ser utilizado para procesar vídeos, en este caso modificamos un poco la instrucción de ejecución para indicarle que debe trabajar sobre un vídeo.

```
./darknet detector demo cfg/voc.data cfg/yolo-obj.cfg yolov3.weights
<video file>
```

Para realizar la predicción sobre vídeos el comando es similar al de imágenes, lo único que cambia es que utilizamos `detector demo` en lugar de `detect`, y le pasamos el path del vídeo en vez del de la imagen. Si el ordenador dispone de cámara web, también es posible trabajar con ella utilizando el mismo comando pero sin indicarle la ruta al vídeo.

Hay que tener en cuenta que en todos los ejemplos anteriores YOLO solo muestra aquellas detecciones para las cuales tiene un nivel de confianza igual o superior al 25%, pero este parámetro también se puede modificar y mostrar detecciones basándonos en un cierto nivel de confianza usando la siguiente instrucción.

```
./darknet detect cfg/yolo-obj.cfg yolov3.weights data/galaygatito.jpg
-thresh .85
```

El único parámetro que se ha añadido es `-thresh .85` donde le indicamos a partir de qué nivel de confianza queremos que detecte y clasifique.

Veamos ejemplos de modificaciones en el nivel de confianza en las siguientes figuras:

- **Ejemplo 1**: Ejecutamos YOLO con un nivel de confianza por defecto, es decir, detectará aquellos objetos que tengan un nivel igual o superior al 25%. Ejecutamos el

siguiente comando:

```
./darknet detect cfg/yolo-obj.cfg yolov3.weights data/galaygatito.jpg
```

Para este nivel de confianza YOLO detecta que hay cuatro objetos: un sofá con un 72% de nivel de confianza, un perro con 89%, un gato con un 97% y una persona con un 52%. Ver la imagen resultante en la figura 31.

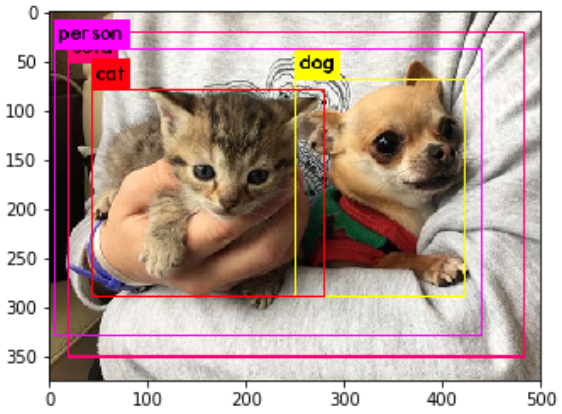


Figura 31. Predicciones de YOLO con un nivel de confianza igual o superior al 25%.

- **Ejemplo 2**: Ejecutamos YOLO con un nivel de confianza del 85%, es decir, solo detectará aquellos objetos que tengan un nivel igual o superior al 85%. Ejecutamos el siguiente comando:

```
./darknet detect cfg/yolo-obj.cfg yolov3.weights data/galaygatito.jpg
-thresh .85
```

Para este nivel de confianza YOLO detecta dos objetos: un perro con 89% y un gato con un 97%. Ver la imagen resultante en la figura 32.

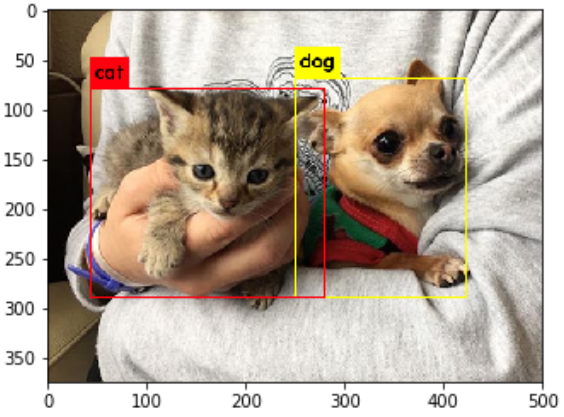


Figura 32. Predicciones de YOLO con un nivel de confianza igual o superior al 85%.

- **Ejemplo 3**: Ejecutamos YOLO con un nivel de confianza del 10%, solo detectará aquellos objetos con un nivel igual o superior al 10%. Ejecutamos el siguiente comando:

```
./darknet detect cfg/yolo-obj.cfg yolov3.weights data/galaygatito.jpg
-thresh .10
```

Para este nivel de confianza YOLO detecta varios objetos: un sofá, una cama, un perro, una persona, otro perro, un gato y una persona. Ver la imagen resultante en la figura 33.

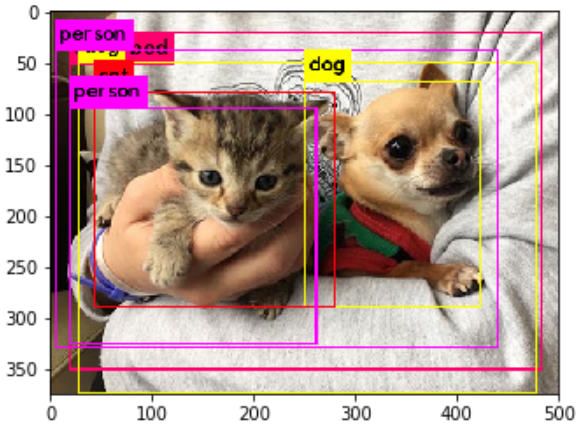


Figura 33. Predicciones de YOLO con un nivel de confianza igual o superior al 10%.

Vemos que los resultados que nos muestra YOLO depende del nivel de confianza que le indiquemos. YOLO detectará más o menos objetos y será más o menos fiable dependiendo de dicho valor.

5. Casos de estudio

Una vez que hemos explicado en la sección anterior los pasos que hay que dar para entrenar y usar un modelo de detección basado en la red YOLO, pasamos a crear un asistente que nos guíe en la creación de este tipo de proyectos. Para ello, primero vamos a ver cual es el entorno de ejecución con el que se va a trabajar para poder ejecutar y entrenar la red YOLO con distintas configuraciones. A continuación, se estudian distintos datasets, en concreto veremos cómo se configura y comportan modelos basados en la red YOLO con el dataset de Pascal VOC, con el de CoCo (del que ya hemos hecho uso en anteriores notebooks), y por último con un dataset propio de imágenes de estomas. Para cada uno de esos datasets generamos un notebook de Jupyter. Por último, a partir de la experiencia obtenida con dichos datasets, terminaremos desarrollando un notebook genérico que se pueda aplicar a cualquier dataset, y que permita crear de manera sencilla modelos de detección de objetos basados en YOLO. Todos los notebooks están disponibles en https://github.com/ancasag/YOLONotebooks

5.1. Entorno de ejecución

El hardware que se va a utilizar es un servidor con la gráfica Nvidia Titan XP. El entorno de ejecución en el que vamos a trabajar tiene de sistema operativo a Ubuntu 16.04, y en él están instaladas las librerías de OpenCV y CUDA, y también Python.

El primer paso para preparar el entorno de ejecución es descargar la librería darknet (una librería *open-source* de redes neuronales) que proporciona la implementación de YOLO. Para eso hacemos uso del siguiente comando:

```
git clone https://github.com/AlexeyAB/darknet
```

Una vez realizado esto ya tenemos la librería en nuestro entorno de trabajo pero es necesario compilarla para que funcione.

Para poder construir modelos basados en la red YOLO tenemos que estar dentro de la carpeta darknet, también tenemos que tener en cuenta que por defecto la librería darknet no se compila con CUDA, ni con OpenCV. Esto puede provocar una serie de problemas ya que al no usar CUDA el procesado es muy lento, y sin OpenCV no se podrán mostrar los resultados. Para evitar esto hay que modificar el fichero `Makefile` antes de compilar la librería. Los cambios a realizar vienen explicados en los notebooks. Una vez listo el entorno, pasamos a explicar cómo se pueden construir modelos de detección basados en YOLO con distintos datasets.

5.2. Dataset Pascal VOC

El dataset Pascal VOC [45] es un proyecto muy popular diseñado con el fin de crear y evaluar algoritmos para la clasificación de imágenes, la detección de objetos y la segmentación. Además del dataset, el proyecto Pascal VOC proporciona herramientas que permiten, por un lado, acceder a los conjuntos de datos y anotaciones; y, por otro, evaluar y comparar diferentes métodos. Este dataset cuenta con un total de 54.000 imágenes que contienen objetos de 20 clases distintas divididas en cuatro grupos: persona, animales, vehículos e interior. Estos cuatro grupos se dividen en las siguientes clases:

- Persona: persona.
- Animal: pájaro, gato, vaca, perro, caballo y oveja.
- Vehículo: avión, bicicleta, barco, autobús, coche, moto y tren.
- Interior: botella, silla, mesa de comedor, planta en maceta, sofá y tv / monitor.

A continuación se explican los aspectos más relevantes para entrenar y utilizar la red YOLO con el dataset Pascal VOC. Una descripción más detallada aparece en el notebook correspondiente.

5.2.1. Recolección del dataset

Para este apartado vamos a usar las imágenes del dataset Pascal VOC desde 2007 hasta 2012, es decir, de las 54.000 imágenes del dataset se van a usar 21.503 imágenes.

Para obtener estos datos es necesario descargarlos y descomprimirlos. Esto nos genera un directorio llamado VOCdevkit donde se descargan todas las imágenes del dataset Pascal VOC y otras herramientas que nosotros no utilizaremos. Interesa sobre todo remarcar que en ese directorio hay 5 carpetas de imágenes: 2007_train, 2007_val, 2007_test, 2012_train y 2012_val. Y además también hay una carpeta de anotaciones con 5 subcarpetas, una para cada carpeta de imágenes.

5.2.2. Data augmentation

En este caso al tener un dataset de 21.503 imágenes, no ha sido necesario aumentarlo aún más.

5.2.3. Anotación del dataset

El siguiente paso consiste en anotar todas las imágenes, es decir, es necesario que cada imagen tenga un archivo .txt con una línea por cada objeto que nos interese detectar. El dataset Pascal VOC ya viene anotado pero no utiliza el mismo formato que se usa para entrenar modelos basados en YOLO. En concreto, Pascal VOC utiliza un formato propio en XML y que se usa habitualmente para anotar objetos. Por lo tanto es necesario transformar las anotaciones del formato Pascal VOC a anotaciones formato YOLO, para lo cual hay que utilizar un script proporcionado por la librería darknet.

Al ejecutar dicho script se genera para cada una de las carpetas de imágenes 2007_test, 2007_train, 2007_val, 2012_train y 2012_val, un archivo .txt donde está la lista de imágenes que contiene cada una de dichas carpetas. Además, para cada imagen se crea un archivo .txt con una línea por cada objeto a detectar en dicha imagen. Dichos archivos .txt se almacenan dentro del directorio VOCdevkit en la carpeta labels.

5.2.4. Dataset split

Este paso es un poco distinto a lo que se hace habitualmente. Normalmente se tiene un conjunto de imágenes y el usuario es el encargado de partirlas en dos conjuntos, uno de entrenamiento y otro de test. En cambio en los desafíos o competiciones como Pascal VOC se proporcionan explícitamente un conjunto de entrenamiento y un conjunto de test para que todos los participantes del concurso evalúen contra el mismo dataset.

Para este caso, el conjunto de entrenamiento está formado por todas las imágenes salvo el conjunto de test del 2007. YOLO necesita un archivo .txt con todas las imágenes que queremos que entrene, por lo que es necesario crear el archivo que contenga los nombres de todas las imágenes que vamos a usar para el entrenamiento.

Por lo que ahora tenemos una lista con todas las imágenes con las que vamos a entrenar en un fichero llamado train.txt, y otra lista con las imágenes para evaluar en un fichero llamado 2007_test.txt. En concreto se usan 16.551 imágenes para el entrenamiento y 4.952 para realizar el test.

5.2.5. Entrenamiento

Como explicamos en la sección 4.5, para entrenar un modelo de YOLO hay que configurar un par de ficheros y ejecutar un comando que inicia el entrenamiento. En este caso, para entrenar dicho modelo no se parte de cero sino que se utilizan los pesos pre-entrenados comentados en la sección 4.5.

5.2.6. Evaluación

Una vez finalizado el entrenamiento, y antes de comprobar que el modelo funciona en imágenes reales, vamos a evaluar el modelo construido en el conjunto de test. Como entrenar un modelo para Pascal VOC lleva mucho tiempo vamos a coger los pesos ya entrenados que son proporcionados por el usuario de github AlexeyAB [63]. Para realizar la evaluación hace falta configurar ciertos ficheros, los detalles están explicados en el notebook. El resultado de la evaluación se muestra en la figura 34, y nos indica el valor de AP para cada clase del dataset, y el valor global de mAP, lo que nos permite clasificarlo como aceptable o no. En este caso un mAP del 74,84% está bastante bien sobre todo teniendo en cuenta que no hay ningún modelo que supere el 78% [64].

```
detections_count = 125598, unique_truth_count = 12032
class_id = 0, name = aeroplane,          ap = 76.98 %
class_id = 1, name = bicycle,    ap = 80.90 %
class_id = 2, name = bird,       ap = 75.46 %
class_id = 3, name = boat,       ap = 64.13 %
class_id = 4, name = bottle,     ap = 48.55 %
class_id = 5, name = bus,        ap = 82.04 %
class_id = 6, name = car,        ap = 82.63 %
class_id = 7, name = cat,        ap = 87.57 %
class_id = 8, name = chair,      ap = 55.71 %
class_id = 9, name = cow,        ap = 80.01 %
class_id = 10, name = diningtable,       ap = 72.47 %
class_id = 11, name = dog,       ap = 85.94 %
class_id = 12, name = horse,     ap = 85.99 %
class_id = 13, name = motorbike,         ap = 83.76 %
class_id = 14, name = person,    ap = 75.92 %
class_id = 15, name = pottedplant,       ap = 50.38 %
class_id = 16, name = sheep,     ap = 77.98 %
class_id = 17, name = sofa,      ap = 71.95 %
class_id = 18, name = train,     ap = 85.13 %
class_id = 19, name = tvmonitor,         ap = 73.22 %
 for thresh = 0.25, precision = 0.68, recall = 0.77, F1-score = 0.72
 for thresh = 0.25, TP = 9224, FP = 4438, FN = 2808, average IoU = 52.45 %

 mean average precision (mAP) = 0.748369, or 74.84 %
Total Detection Time: 49.000000 Seconds
```

Figura 34. Proceso de evaluación con el modelo de Pascal VOC.

5.2.7. Predicción

Por último probamos nuestro modelo con imágenes que no se encuentran ni en el conjunto de entrenamiento ni en el de test. Vemos que ha detectado que en la siguiente imagen hay un perro, un coche y una bicicleta:

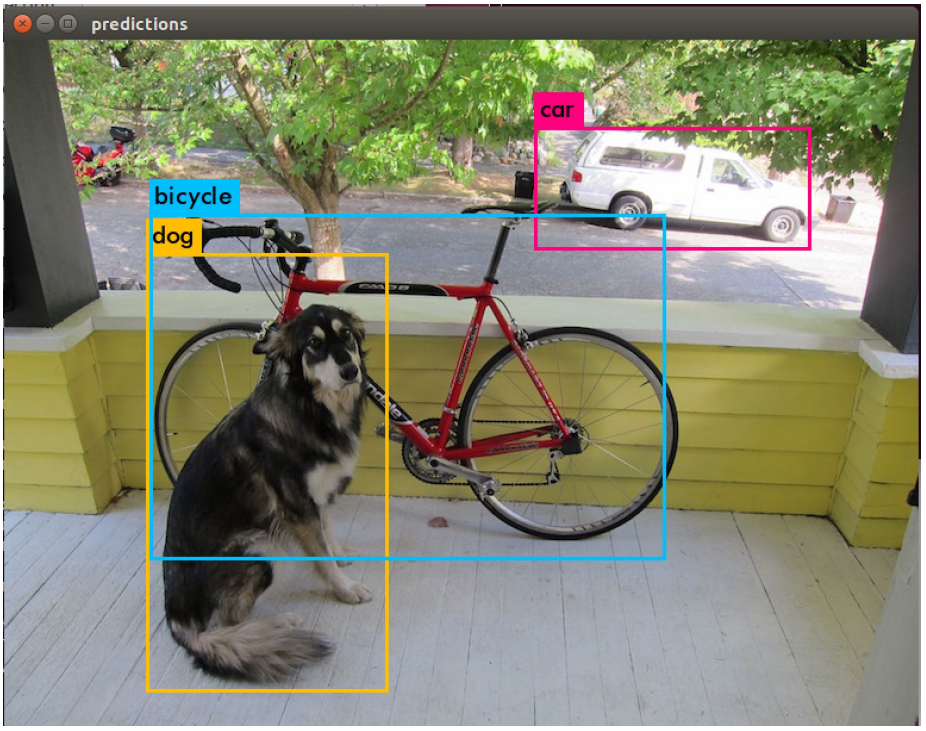


Figura 35. Proceso de predicción con el modelo de Pascal VOC.

También se han hecho pruebas con vídeos, y el resultado se puede consultar en el notebook.

5.3. Dataset CoCo

El siguiente dataset que consideramos es CoCo [55]. Es un dataset diseñado con el fin de avanzar en la detección de objetos, segmentación, detección de puntos clave de personas, segmentación de elementos y generación de leyendas. Este paquete proporciona las API de Matlab, Python y Lua que ayudan a cargar, analizar y visualizar las anotaciones en CoCo. Este dataset cuenta con un total de 300.000 imágenes etiquetadas, que contienen objetos de 80 clases distintas, divididas en varios grupos (ver figura 36), como pueden ser personas, gatos, barcos, trenes, perros, etc.

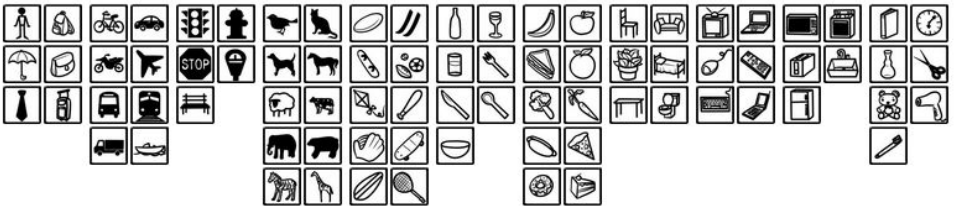


Figura 36. Lista de clases del dataset CoCo.

Al igual que con Pascal VOC pasamos a comentar los aspectos más importantes para entrenar y utilizar este dataset. Para más detalles ver el notebook correspondiente.

5.3.1. Recolección del dataset

En este caso el dataset está formado por las imágenes de la base de datos de CoCo desde 2015 hasta 2018, es decir, de las 300.000 imágenes del dataset se van a usar 122.260 imágenes. Para obtener estos datos es necesario descargarlos y descomprimirlos. Interesa sobre todo remarcar que al descargar el dataset se generan 2 carpetas de imágenes: train2014 y val2014. Además también hay una carpeta que contiene las imágenes anotadas en el formato que requiere la librería.

5.3.2. Data augmentation

En este caso al tener un dataset de 122.260 imágenes, no ha sido necesario aumentarlo aún más.

5.3.3. Anotación del dataset

Como hemos comentado anteriormente, al descargar el dataset de CoCO también se descargan las anotaciones en el formato que necesita la librería, por lo que no hay que hacer nada más en este paso.

5.3.4. Dataset split

Al igual que como ocurría con Pascal VOC, CoCo tiene preparados conjuntos para que no sea necesario hacer particiones.

Para este caso el conjunto de entrenamiento serán las imágenes contenidas en la carpeta train2014, que contiene 117.264 imágenes, y el de test serán las imágenes contenidas en la carpeta val2014, que contiene 4.996 imágenes.

5.3.5. Entrenamiento

No hay diferencias significativas con el proceso realizado para el dataset de Pascal VOC.

5.3.6. Evaluación

El proceso es el mismo explicado para Pascal VOC. El mAP obtenido para este dataset es inferior que para el de Pascal VOC (ver figura 37), esto es debido a que el dataset tiene mayor complejidad (más imágenes y clases). Aunque un 54,37% pueda parecer un mAP bajo, el mejor modelo para este dataset solo ha alcanzado un 59,1% [65].

```
class_id = 47, name = apple,      ap = 20.18 %
class_id = 48, name = sandwich,           ap = 51.30 %
class_id = 49, name = orange,     ap = 34.27 %
class_id = 50, name = broccoli,           ap = 33.79 %
class_id = 51, name = carrot,     ap = 25.56 %
class_id = 52, name = hot dog,    ap = 43.01 %
class_id = 53, name = pizza,      ap = 59.55 %
class_id = 54, name = donut,      ap = 45.72 %
class_id = 55, name = cake,       ap = 50.17 %
class_id = 56, name = chair,      ap = 44.07 %
class_id = 57, name = sofa,       ap = 59.58 %
class_id = 58, name = pottedplant,        ap = 44.48 %
class_id = 59, name = bed,        ap = 67.93 %
class_id = 60, name = diningtable,        ap = 46.87 %
class_id = 61, name = toilet,     ap = 75.49 %
class_id = 62, name = tvmonitor,          ap = 74.30 %
class_id = 63, name = laptop,     ap = 70.49 %
class_id = 64, name = mouse,      ap = 71.63 %
class_id = 65, name = remote,     ap = 48.55 %
class_id = 66, name = keyboard,           ap = 67.07 %
class_id = 67, name = cell phone,         ap = 43.15 %
class_id = 68, name = microwave,          ap = 70.85 %
class_id = 69, name = oven,       ap = 51.26 %
class_id = 70, name = toaster,    ap = 17.49 %
class_id = 71, name = sink,       ap = 59.61 %
class_id = 72, name = refrigerator,       ap = 72.04 %
class_id = 73, name = book,       ap = 17.17 %
class_id = 74, name = clock,      ap = 72.56 %
class_id = 75, name = vase,       ap = 51.57 %
class_id = 76, name = scissors,           ap = 39.51 %
class_id = 77, name = teddy bear,         ap = 59.71 %
class_id = 78, name = hair drier,         ap = 9.48 %
class_id = 79, name = toothbrush,         ap = 36.30 %
 for thresh = 0.25, precision = 0.64, recall = 0.54, F1-score = 0.59
 for thresh = 0.25, TP = 19313, FP = 10865, FN = 16444, average IoU = 50.17 %

 mean average precision (mAP) = 0.543687, or 54.37 %
Total Detection Time: 100.000000 Seconds
```

Figura 37. Proceso de evaluación de YOLO con el modelo de CoCo.

5.3.7. Predicción

Por último probamos este modelo con imágenes que no se encuentran ni en el conjunto de entrenamiento ni en el de test. Vemos que ha detectado que en la siguiente imagen hay un perro, un camión y una bicicleta.

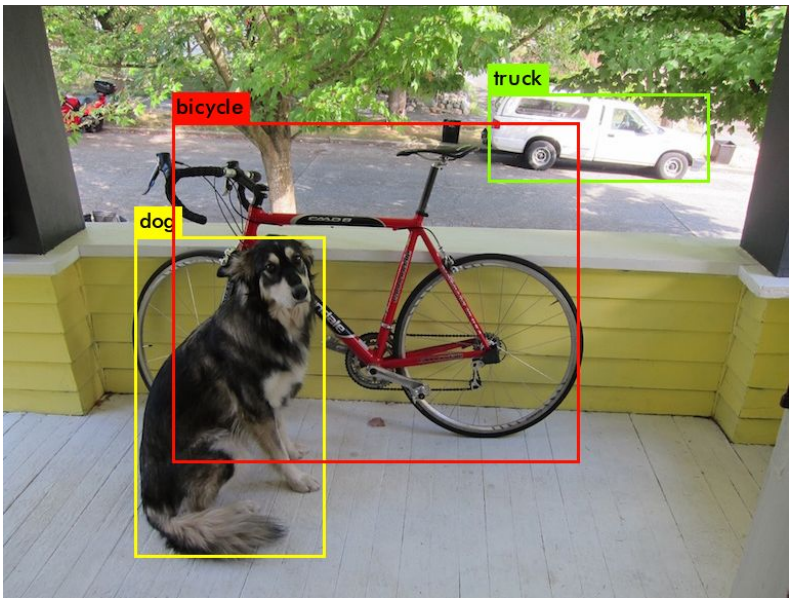


Figura 38. Proceso de predicción de YOLO con el modelo de CoCo.

Hasta ahora hemos visto cómo se puede entrenar YOLO con dos datasets que están, por así decirlo, preparados para ser utilizados por este modelo. A continuación pasamos a ver cómo entrenamos YOLO con un dataset propio.

5.4. Dataset de estomas

Para esta sección se va a utilizar un dataset de estomas, que representa un caso real de cómo se puede utilizar un modelo basado en la red YOLO para resolver el problema de detección en un dataset propio. Los estomas son poros o aberturas regulables del tejido epidérmico de las plantas (ver figura 39), formados por un par de células especializadas, denominadas células oclusivas o guarda. Al poro en sí, se le denomina ostiolo, que comunica hacia el interior con una cavidad denominada cámara subestomática. Adyacente a cada célula guarda se encuentran generalmente 1 o 2 células epidérmicas modificadas que reciben el nombre de células subsidiarias o accesorias, siendo las células oclusivas las que controlan la apertura de los estomas.

Figura 39. Ejemplo de estoma.

El número y comportamiento de los estomas proporciona información clave para medir los niveles de estrés de agua, ratio de producción, y la salud general de la planta [66]. Por lo tanto, analizar el número y comportamiento de los estomas puede servir, entre otras cosas, para gestionar mejor los cultivos en agricultura [66]. Sin embargo, contar de manera manual

el número de estomas de una imagen es un proceso costoso debido al número de estomas que hay en cada imagen, ver figura 40, y que es un proceso que se debe hacer no solo sobre una imagen sino sobre lotes de decenas de imágenes. Por lo tanto, es interesante ser capaces de automatizar el proceso de detección.

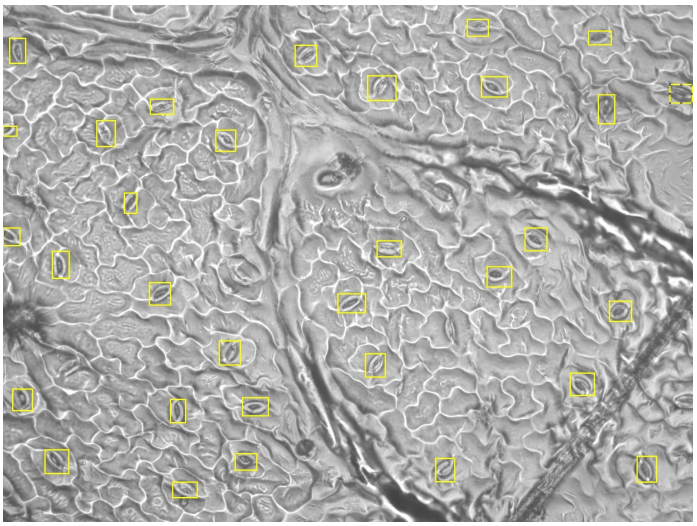


Figura 40. Ejemplo de detección de estomas.

5.4.1. Recolección del dataset

El dataset de estomas ha sido cedido para este proyecto por la Universidad de Missouri a través de la Universidad de La Rioja. El dataset inicial cuenta con 39 imágenes de estomas de tamaño 1600×1200 anotadas en formato Pascal VOC por expertos biólogos utilizando el programa *LabelImg* [58] (el proceso de anotación es costoso, lo que explica el número limitado de imágenes).

Las imágenes de tamaño 1600×1200 son demasiado grandes para entrenar la red YOLO (no es factible cargar en memoria imágenes tan grandes), por lo que dichas imágenes se han partido en subimágenes de tamaño 400×400, obteniendo un total de 468 imágenes, con un total de 1652 estomas, que mantienen su anotación correspondiente.

5.4.2. Data augmentation

Este es un dataset pequeño, ya que como hemos dicho en el apartado anterior, solo cuenta con 468 imágenes, por lo que es necesario aumentarlo aplicando al conjunto inicial de imágenes una serie de filtros, rotaciones, etc. Para esto se ha usado una librería desarrollada en la Universidad de La Rioja, llamada CLoDSA [67]. Esta librería permite aplicar, a un conjunto de imágenes anotadas, una serie de transformaciones y obtener, de manera automática, las imágenes transformadas y las anotaciones correspondientes. De este modo, el usuario solo tiene que anotar el conjunto de imágenes inicial, y la librería se encarga de manera automática de generar las imágenes aumentadas y sus correspondientes anotaciones, evitando de este modo la anotación manual de muchas imágenes. Para este dataset se han aplicado transformaciones que producen imágenes que siguen teniendo

sentido; es decir, que un biólogo reconocería como imágenes de estomas; por ejemplo, no tendría sentido en este caso realizar una transformación que distorsione la imagen. En concreto, se han aplicado las siguientes 8 técnicas:

- Volteo vertical.
- Volteo horizontal.
- Volteo horizontal seguido de un volteo vertical.
- Rotaciones de 90, 180 y 270 grados.
- Filtro de la media.
- Ruido Gaussiano.

Cada una de estas transformaciones se aplica al conjunto inicial, por lo que el dataset inicial se multiplica por 8. Este proceso para aumentar las imágenes ha sido documentado en un notebook de Jupyter. Al final del proceso de *data augmentation* contamos con un total de 4212 imágenes en el dataset (las 468 originales y las 3744 generadas por CLoDSA) que ya es un número suficiente para entrenar nuestro modelo.

5.4.3. Anotación del dataset

Como hemos comentado antes, el dataset de los estomas ha sido anotado utilizando el formato Pascal VOC, por lo que es necesario convertir dichas anotaciones al formato YOLO. Para eso se usa un script de Python que realiza dicha tarea.

5.4.4. Dataset split

El siguiente paso consiste en dividir el dataset en dos conjuntos basándonos en la división aproximada 90-10%, por lo que tendremos dos conjuntos, uno de 3744 imágenes (el de entrenamiento) y otro de 468 imágenes (el de test). Como hemos comentado anteriormente, la librería necesita un archivo .txt donde tenga listadas las imágenes que se van a usar en el entrenamiento y lo mismo en el caso de las de test, estos archivos son test.txt para las imágenes de test y train.txt para las de entrenamiento (estos ficheros se han generado en el apartado anterior al ejecutar el script del apartado anterior). Además, las imágenes y anotaciones deben estar organizadas en carpetas de una manera concreta (ver el notebook), de lo contrario el proceso de entrenamiento falla.

5.4.5. Entrenamiento

Como explicamos en la sección 4.5, para entrenar un modelo de YOLO hay que configurar unos ficheros y ejecutar un comando que inicie el entrenamiento. Para la configuración de dichos ficheros se han tomado como base los ficheros utilizados para entrenar el dataset de CoCo, y se han realizado los siguientes cambios:

- `cfg/vocEstomas.data`: se indica la ruta a las imágenes y se especifica que hay una única clase.
- `data/vocEstomas.names`: se proporciona la lista de clases del dataset que en este caso es únicamente “estoma”.

- `cfg/yolov3.cfg`: se modifica el valor de dos parámetros, el parámetro *classes* (donde indicamos el número de clases de nuestro dataset, en este caso 1) y el parámetro *filters* (al que se le asigna el valor que corresponde a este operación, filters=(classes + 5)x3, por lo que en este caso será filters= (1+5)x3 = 18).

El resto del proceso para entrenar el modelo es análogo al seguido en el resto de datasets. Como en casos anteriores, para entrenar dicho modelo no se parte de cero sino que se utilizan los pesos pre-entrenados comentados en la sección 4.5. Aunque se ha entrenado el modelo por completo durante 250.000 épocas, para decidir con qué modelo nos vamos a quedar se evalúan los pesos generados en las iteraciones 220.000, 230.000, 240.000, y por último, la 250.000.

5.4.6. Evaluación

Una vez finalizado el entrenamiento, y antes de poner el modelo en producción, vamos a evaluar el modelo construido en el conjunto de test. El proceso para realizar la evaluación es el mismo que para los otros casos, pero en esta ocasión contamos con varios modelos (que vienen dados por los distintos pesos obtenidos durante la fase de entrenamiento) y que por lo tanto hay que evaluarlos. Cuando contamos con varios modelos, es importante no quedarnos solo con el último generado ya que, como dijimos en la sección 4.6, el último fichero .weigth que se ha obtenido no tiene porque ser el que mejor resultado nos de, ya que se puede producir un sobreentrenamiento. Por lo tanto, es necesario realizar una comparación sobre los 4 o 5 últimos modelos que se han generado en el entrenamiento.

Como hemos dicho anteriormente, evaluamos los modelos obtenidos en las épocas 220.000, 230.000, 240.000 y 250.000. Como puede verse en la tabla 3, todos los modelos tienen el mismo valor de mAP (un 90,9%), por lo que necesitamos fijarnos en otros parámetros como pueden ser la *precision*, el *F1-score* y el valor del IOU.

	Precision					*F1-score*				
Threshold / Pesos	0,1	0,25	0,5	0,75	0,9	0,1	0,25	0,5	0,75	0,9
220000	0,95	0,98	0,99	1	1	0,97	0,98	0,99	0,98	0,92
230000	0,97	0,99	0,99	1	1	0,98	0,99	0,99	0,99	0,98
240000	0,97	0,98	0,99	1	1	0,98	0,98	0,98	0,98	0,97
250000	0,98	0,98	0,99	1	1	0,98	0,99	0,99	0,98	0,97

Tabla 3. Resultado de la evaluación de modelos con diferentes pesos y valores de threshold.

Fijándonos en estos parámetros vemos que el modelo con mejores resultados es el modelo yolov3_250000 (el último construido) con un *threshold* de 0,5; ver figura 41 (el resto de resultados puede verse en el notebook correspondiente).

```
detections_count = 1823, unique_truth_count = 1638
class_id = 0, name = estoma,      ap = 90.91 %
 for thresh = 0.75, precision = 1.00, recall = 0.97, F1-score = 0.98
 for thresh = 0.75, TP = 1590, FP = 4, FN = 48, average IoU = 87.16 %

 mean average precision (mAP) = 0.909091, or 90.91 %
Total Detection Time: 8.000000 Seconds
```

Figura 41. Resultado de calcular el mAP con el fichero de pesos yolov3_250000.

Una vez que ya tenemos el modelo que nos da los mejores resultados, nos preguntamos si los resultados obtenidos son lo suficientemente buenos. En el artículo *Microscope image based fully automated stomata detection and pore measurement method for grapevines* [68] se obtiene una *precision* de 91,68% y un *F1-score* de 85% utilizando las técnicas clásicas de detección de objetos, mientras que en nuestro caso se ha obtenido una *precision* del 98%, un *F1-score* del 99%, y además un valor mAP, que como hemos dicho antes, es de 90,91%. Por lo que podemos considerar que los resultados obtenidos con nuestro modelo son bastante positivos. Cabe notar que el trabajo presentado en [68] y el nuestro no utilizan el mismo dataset (ni las imágenes ni el código utilizado en [68] están disponibles), aún así podemos decir que nuestro modelo obtiene resultados comparables a los disponibles en la literatura.

5.4.7. Predicción

Por último probamos nuestro modelo con imágenes que no se encuentran ni en el conjunto de entrenamiento ni en el de test. Vemos que el modelo ha detectado correctamente que en la siguiente imagen hay varios estomas.

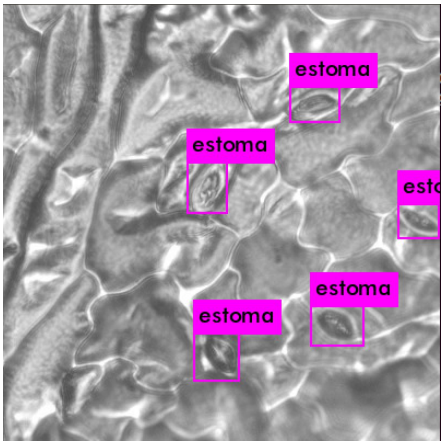


Figura 42. Resultado de predicción en una imagen dividida.

Aunque el dataset que se utilizó para entrenar usaba imágenes de tamaño 400×400, al modelo entrenado se le pueden pasar imágenes completas, sin necesidad de recortarlas (ver figura 43) y el modelo es capaz de trabajar con ellas sin problema.

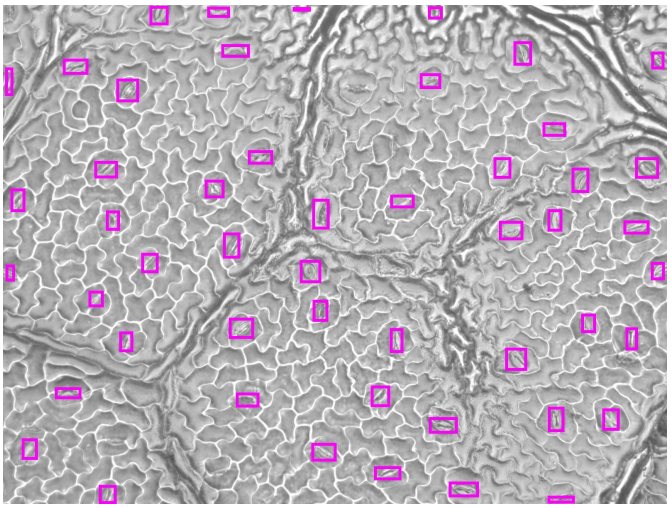


Figura 43. Resultado de predicción de una imagen completa.

Como hemos visto en este apartado, la red YOLO puede utilizarse de manera exitosa para crear modelos de detección en datasets de imágenes propios, así que queda más que justificado crear una herramienta para que usuarios no expertos puedan beneficiarse de ella.

5.5. Notebook genérico

Como hemos visto, para entrenar un modelo basado en YOLO hay que tener mucho cuidado, ya que la librería es muy sensible a la estructura de carpetas y a todos los ficheros de configuración, por lo que es un proceso propenso a cometer errores. Por lo tanto, una vez adquirida la experiencia necesaria para trabajar con modelos basados en YOLO usando distintos datasets, se ha creado un notebook genérico que automatiza la mayor parte del proceso de creación de modelos, evita errores, y puede ser usado por cualquier usuario para crear sus propios modelos.

Usando dicho notebook genérico, el usuario solo tiene que realizar las siguientes dos operaciones: (1) crear una carpeta con las imágenes y sus correspondientes anotaciones; y (2) definir cuatro parámetros en el notebook: el nombre del proyecto, la ruta a la carpeta con las imágenes y las anotaciones, la lista de las clases a detectar y el porcentaje de imágenes usado para entrenar.

Una vez hecho esto, el notebook se encarga de manera automática, mediante una serie de funciones implementadas en Python, de:

1. Validar que la carpeta contiene las imágenes en el formato adecuado.
2. Validar que todas las imágenes tienen asociada su anotación.
3. Contar el número de clases a detectar.
4. Generar el fichero de clases.
5. Indicar al usuario cómo aumentar el dataset.
6. Realizar la separación de los datos en dos conjuntos, el de entrenamiento y el de test, basándose en el porcentaje proporcionado por el usuario.

7. Generar los ficheros train.txt y test.txt, donde como sabemos de secciones anteriores, tenemos almacenadas las rutas de las imágenes de los conjuntos de entrenamiento y de test.
8. Generar el fichero .data para indicar dónde, se encuentran los ficheros anteriores.
9. Generar el fichero donde indicamos cómo se va a realizar el entrenamiento.
10. Generar la instrucción de entrenamiento, para que el usuario pueda ejecutarla simplemente copiando y pegándola.
11. Generar el fichero donde se indica cómo se va a realizar la evaluación.
12. Listar los pesos que se han generado en el proceso de entrenamiento, permitiendo al usuario elegir entre ellos para poder evaluarlos.
13. Crear la instrucción de evaluación, para que el usuario pueda ejecutarla.
14. Por último, crear la instrucción de predicción del modelo sobre una imagen que no pertenezca al conjunto de datos inicial.

El correcto funcionamiento del notebook se ha probado replicando el trabajo del dataset de los estomas.

Notar que los 14 pasos anteriores son obligatorios si queremos crear un modelo basado en YOLO, y llevarlos a cabo de manera manual es un proceso muy costoso, y en el cual es muy sencillo cometer algún error. Por lo tanto, gracias a este trabajo se consigue facilitar muchísimo el proceso de creación de modelos basados en la red YOLO y cualquier usuario, experto o no experto, puede beneficiarse de esta técnica.

6. Conclusiones

El problema que se abordaba en este trabajo era facilitar a usuarios no expertos la construcción de modelos de detección de objetos en imágenes y vídeos. Este proceso consta de muchos pasos, donde cada uno de ellos depende del anterior, lo cual exige seguir un orden muy concreto. Además tiene otro inconveniente y es que cada paso debe ser realizado por una persona especializada en dicha tarea, lo que quiere decir que no puede ser realizado por una persona inexperta.

En este trabajo se ha desarrollado una herramienta (mediante cuadernos de Jupyter) que guía a usuarios no expertos a la hora de construir modelos de detección. Para construir dicha herramienta ha sido necesario aprender diversos conceptos sobre aprendizaje automático, en concreto técnicas de *deep learning*, análisis de imagen y detección de objetos. La herramienta permite construir modelos de detección que utilizan el enfoque de YOLO, ya que como hemos visto es uno de los métodos más efectivos para la detección de objetos. Además de facilitar la construcción de modelos de detección, los cuadernos tienen como valor añadido que son materiales de enseñanza ya que se han incluido explicaciones, tanto textuales como gráficas, de todos los pasos involucrados en la creación del modelo, así como de distintos conceptos de análisis de imagen y aprendizaje automático.

Ya que el mayor inconveniente de este trabajo es el hecho de que los usuarios necesiten una GPU (ya que, en caso contrario no es factible crear modelos de detección) se plantea, en caso de continuar con este trabajo, integrar la herramienta en un servicio en la nube como Amazon o Google Cloud. Además de ampliar a otras técnicas de *deep learning* para la detección de objetos como SSD.

Este trabajo se enmarca dentro del Máster en Tecnologías Informáticas, y las asignaturas de dicho máster han aportado una gran cantidad de conocimientos y herramientas que han facilitado el desarrollo de este trabajo, entre las que podemos destacar:

- *Tecnologías semánticas*: en esta asignatura se nos enseñó el funcionamiento de los cuadernos de Jupyter, y haciendo uso de ellos hemos podido desarrollar toda la implementación del trabajo. Además en esta asignatura se veían diversos conceptos sobre aprendizaje automático que han servido como base para este TFM.
- *Procesamiento de imágenes digitales*: donde se ha visto la aplicación de filtros en imágenes y nos ha permitido entender las técnicas clásicas de detección de objetos.
- *Modelización y tratamiento científico de datos*: hemos visto la diferencia de utilizar la GPU frente a la CPU, lo que nos ha permitido crear modelos más rápidos.

El desarrollo de este proyecto ha sido sumamente interesante, además de muy satisfactorio, ya que me ha permitido adentrarme en la inteligencia artificial, en el tratamiento de imágenes, en el deep learning y en concreto, en el uso de redes neuronales profundas.

Bibliografía

1. Mata Martínez G. Processing biomedial images for the study of treatments related to neurodegenerative diseases. Universidad de La Rioja. 2017.

2. Heras J, Domínguez C, Mata E, Pascual V, Lozano C, Torres C, et al. GelJ – a tool for analyzing DNA fingerprint gel images. BMC Bioinformatics. 2015;16. doi:10.1186/s12859-015-0703-0

3. Alonso CA, Domínguez C, Heras J, Mata E, Pascual V, Torres C, et al. Antibiogramj: A tool for analysing images from disk diffusion tests. Comput Methods Programs Biomed. 2017;143: 159–169.

4. Kluyver T. Jupyter Notebooks—a publishing format for reproducible computational workflows. 20th International Conference on Electronic Publishing.

5. FAICO. ¿De dónde viene la Visión Artificial? [Internet]. [cited 6 Mar 2018]. Available: http://faicoblog.blogspot.com/2016/02/de-donde-viene-la-vision-artificial.html

6. Salza C. Los tres grandes problemas que tiene la Inteligencia Artificial para evolucionar [Internet]. [cited 19 Jun 2018]. Available: https://prnoticias.com/tecnologia/20156823-inteligencia-artificial-problemas

7. Szegedy C, Liu W, Jia Y, Sermanet P, Reed S, Anguelov D, et al. Going deeper with convolutions. 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). 2015. doi:10.1109/cvpr.2015.7298594

8. He K, Zhang X, Ren S, Sun J. Deep Residual Learning for Image Recognition. 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). 2016. doi:10.1109/cvpr.2016.90

9. Agmon N, Moseley R, Nwamba C, Bendell C, Shiran A, Otemuyiwa P, et al. Cloudinary Blog [Internet]. [cited 19 Jun 2018]. Available: https://cloudinary.com/blog/

10. Hussin R, Rizon Juhari M, Kang NW, Ismail RC, Kamarudin A. Digital Image Processing Techniques for Object Detection From Complex Background Image. Procedia Engineering. 2012;41: 340–344.

11. Lopez Briega RE. Machine Learning con Python [Internet]. [cited 19 Jun 2018]. Available: http://relopezbriega.github.io/blog/2015/10/10/machine-learning-con-python/

12. ¿Qué es el Machine Learning? In: Jarroba [Internet]. 30 Jan 2016 [cited 19 Jun 2018]. Available: http://jarroba.com/que-es-el-machine-learning/

13. Aprendizaje automático - Wikipedia, la enciclopedia libre [Internet]. [cited 19 Jun 2018]. Available: https://es.wikipedia.org/wiki/Aprendizaje_autom%C3%A1tico

14. González A. Conceptos básicos de Machine Learning - [Internet]. 30 Jul 2014 [cited 19 Jun 2018]. Available: http://cleverdata.io/conceptos-basicos-machine-learning/

15. Caparrini FS, Work WW. Introducción al Aprendizaje Automático - Fernando Sancho

Caparrini [Internet]. 23 Sep 2017 [cited 19 Jun 2018]. Available: http://www.cs.us.es/~fsancho/?e=75

16. Montes MC. K-Nearest Neighbors Gráficas, estadística y minería de datos con python. In: CIEMAT [Internet]. [cited 19 Jun 2018]. Available: http://wwwae.ciemat.es/~cardenas/docs/curso_MD/knn.pdf

17. Cárdenas-Montes M. RANDOM FOREST. In: CIEMAT [Internet]. [cited 19 Jun 2018]. Available: http://wwwae.ciemat.es/~cardenas/docs/lessons/RandomForest.pdf

18. Betancourt GA. LAS MÁQUINAS DE SOPORTE VECTORIAL (SVMs). 2005; Available: https://dialnet.unirioja.es/descarga/articulo/4838384.pdf

19. Izaurieta F, Saavedra C. Redes neuronales artificiales. Available: http://www.uta.cl/charlas/volumen16/Indice/Ch-csaavedra.pdf

20. Sewell M. No Free Lunch Theorems [Internet]. [cited 19 Jun 2018]. Available: http://www.no-free-lunch.org/

21. Website [Internet]. [cited 19 Jun 2018]. Available: http://medicinaycomplejidad.org/pdf/reciente/r31459.pdf

22. Muñoz Pérez J. El Perceptrón Simple. Available: http://www.lcc.uma.es/~munozp/documentos/modelos_computacionales/temas/Tema4MC-05.pdf

23. Arrabales R. Deep Learning: qué es y por qué va a ser una tecnología clave en el futuro de la inteligencia artificial [Internet]. 29 Mar 2016 [cited 20 Jun 2018]. Available: https://www.xataka.com/robotica-e-ia/deep-learning-que-es-y-por-que-va-a-ser-una-tecnologia-clave-en-el-futuro-de-la-inteligencia-artificial

24. Procesamiento de lenguajes naturales - Wikipedia, la enciclopedia libre [Internet]. [cited 20 Jun 2018]. Available: https://es.wikipedia.org/wiki/Procesamiento_de_lenguajes_naturales

25. Visión artificial - Wikipedia, la enciclopedia libre [Internet]. [cited 20 Jun 2018]. Available: https://es.wikipedia.org/wiki/Visi%C3%B3n_artificial

26. Local binary patterns - Wikipedia [Internet]. [cited 20 Jun 2018]. Available: https://en.wikipedia.org/wiki/Local_binary_patterns

27. Haar-like feature - Wikipedia [Internet]. [cited 20 Jun 2018]. Available: https://en.wikipedia.org/wiki/Haar-like_features#cite_note-2

28. van Veen F. The Neural Network Zoo - The Asimov Institute. In: The Asimov Institute [Internet]. 14 Sep 2016 [cited 20 Jun 2018]. Available: http://www.asimovinstitute.org/neural-network-zoo/

29. Manger R, Puljic K. Multilayer Perceptrons and Data Compression. 2007;26: 45–62.

30. Redes Neuronales [Internet]. [cited 20 Jun 2018]. Available: https://ml4a.github.io/ml4a/es/neural_networks/

31. Valencia Reyes MA, Yañéz Marquéz C, & Sánchez Fernández LP. [PDF]Algoritmo Backpropagation para Redes Neuronales: conceptos y . 2006; Available:

http://repositoriodigital.ipn.mx/bitstream/123456789/8628/1/Archivo%20que%20incluye%20portada,%20%C3%ADndice%20y%20texto.pdf

32. Fukushima K. Neocognitron: a self organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. Biol Cybern. 1980;36: 193–202.

33. Lecun Y, Bottou L, Bengio Y, Haffner P. Gradient-based learning applied to document recognition. Proc IEEE. 1998;86: 2278–2324.

34. Ericson G, Toliver K, Takaki J, Petersen T, Martens J. Data Transformation - Filter - Azure Machine Learning Studio. In: Microsoft Azure [Internet]. [cited 20 Jun 2018]. Available: https://docs.microsoft.com/en-us/azure/machine-learning/studio-module-reference/data-transformation-filter

35. OpenCV: Introduction [Internet]. [cited 20 Jun 2018]. Available: https://docs.opencv.org/master/d1/dfb/intro.html

36. Applications - Keras Documentation [Internet]. [cited 20 Jun 2018]. Available: https://keras.io/applications/#inceptionv3

37. Applications - Keras Documentation [Internet]. [cited 20 Jun 2018]. Available: https://keras.io/applications/#vgg16

38. Applications - Keras Documentation [Internet]. [cited 20 Jun 2018]. Available: https://keras.io/applications/#xception

39. ImageNet [Internet]. [cited 20 Jun 2018]. Available: http://www.image-net.org/

40. Viola P, Jones M. Robust Real-time Object Detection [Internet]. [cited 20 Jun 2018]. Available: https://www.cs.cmu.edu/~efros/courses/LBMV07/Papers/viola-IJCV-01.pdf

41. [PDF]Cap´ıtulo 3 Clasificadores Débiles - AdaBoost. Available: http://catarina.udlap.mx/u_dl_a/tales/documentos/lmt/morales_s_aa/capitulo3.pdf

42. Rosebrock A. Histogram of Oriented Gradients and Object Detection - PyImageSearch. In: PyImageSearch [Internet]. 10 Nov 2014 [cited 20 Jun 2018]. Available: https://www.pyimagesearch.com/2014/11/10/histogram-oriented-gradients-object-detection/

43. Pyramid (image processing) - Wikipedia [Internet]. [cited 20 Jun 2018]. Available: https://en.wikipedia.org/wiki/Pyramid_(image_processing)

44. Hosang J, Benenson R, Schiele B. Learning non-maximum suppression. CoRR. 2017;abs/1705.02950. Available: https://arxiv.org/abs/1705.02950

45. The PASCAL Visual Object Classes Homepage [Internet]. [cited 20 Jun 2018]. Available: http://host.robots.ox.ac.uk/pascal/VOC/

46. Rosebrock A. Intersection over Union (IoU) for object detection - PyImageSearch. In: PyImageSearch [Internet]. 7 Nov 2016 [cited 20 Jun 2018]. Available: https://www.pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-detection/

47. Hui J. mAP (mean Average Precision) for Object Detection – Jonathan Hui – Medium.

In: Medium [Internet]. Medium; 7 Mar 2018 [cited 20 Jun 2018]. Available: https://medium.com/@jonathan_hui/map-mean-average-precision-for-object-detection-45c121a31173

48. Girshick R, Donahue J, Darrell T, Malik J. Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation. Proceedings of the 2014 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'14). 2014. pp. 580–587.

49. Redmon J, Divvala S, Girshick R, Farhadi A. You only look once: Unified, real-time object detection. Proceedings of the IEEE conference on computer vision and pattern recognition. 2016. pp. 779–788.

50. Girshick R. Fast R-CNN. Proceedings of the 2015 IEEE International Conference on Computer Vision (ICCV 2015). 2015. pp. 1080–1088.

51. Ren S, He K, Girshick R, Sun J. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. IEEE Trans Pattern Anal Mach Intell. 2017;39: 1137–1149.

52. Liu W, Others. SSD: Single Shot MultiBox Detector. Proceedings of the 14th European Conference on Computer Vision (ECCV 2016). 2016. pp. 21–37.

53. Brownlee J. How to One Hot Encode Sequence Data in Python. In: Machine Learning Mastery [Internet]. 12 Jul 2017 [cited 20 Jun 2018]. Available: https://machinelearningmastery.com/how-to-one-hot-encode-sequence-data-in-python/

54. Anchor box - Machine Learning Glossary [Internet]. [cited 20 Jun 2018]. Available: https://machinelearning.wtf/terms/anchor-box/

55. Lin T-Y, Maire M, Belongie S, Hays J, Perona P, Ramanan D, et al. Microsoft COCO: Common Objects in Context. In: Fleet D, Pajdla T, Schiele B, Tuytelaars T, editors. Computer Vision -- ECCV 2014. Cham: Springer International Publishing; 2014. pp. 740–755.

56. Wang J, Perez L. The effectiveness of data augmentation in image classification using deep learning [Internet]. Technical report; 2017. Available: http://cs231n.stanford.edu/reports/2017/pdfs/300.pdf

57. FFmpeg [Internet]. [cited 20 Jun 2018]. Available: https://www.ffmpeg.org/

58. Tzutalin D. LabelImg [Internet]. 2015. Available: https://github.com/tzutalin/labelImg

59. Alexey AB. YOLO mark [Internet]. 2018. Available: https://github.com/AlexeyAB/Yolo_mark

60. Weill E. Convert Datasets [Internet]. 2017. Available: https://github.com/eweill/convert-datasets

61. Redmon J. ImageNet Classification [Internet]. [cited 21 Jun 2018]. Available: https://pjreddie.com/darknet/imagenet/#darknet53

62. Redmon J. YOLO: Real-Time Object Detection [Internet]. [cited 21 Jun 2018]. Available: https://pjreddie.com/darknet/yolo/

63. AlexeyAB. AlexeyAB/darknet. In: GitHub [Internet]. [cited 21 Jun 2018]. Available: https://github.com/AlexeyAB/darknet

64. Redmon J, Farhadi A. YOLO9000: Better, Faster, Stronger. 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). 2017. doi:10.1109/cvpr.2017.690

65. Lin T-Y, Others. Feature Pyramid Networks for Object Detection. CoRR. 2016; Available: https://arxiv.org/abs/1612.03144

66. Buttery BR, Tan CS, Buzzell RI, Gaynor JD, MacTavish DC. Stomatal numbers of soybean and response to water stress. Plant Soil. 1993;149: 283–288.

67. Heras J, Others. CLoDSA: an open-source image augmentation library for object classification, localization, detection and semantic segmentation [Internet]. 2018. Available: https://github.com/joheras/CLoDSA

68. Jayakody H, Liu S, Whitty M, Petrie P. Microscope image based fully automated stomata detection and pore measurement method for grapevines. Plant Methods. 2017;13. doi:10.1186/s13007-017-0244-9

Anexo 1: Notebooks de Jupyter

Aquí nos podemos encontrar con un conjunto de Noteboks de Jupyter que se corresponden con la implementación del Trabajo de Fin de Máster:

- **Filtros.ipynb**: en este notebook se muestra cómo funciona el operador de convolución y cómo los filtros nos van a permitir, entre otras cosas, resaltar los bordes de la imagen para su detección.
- **VentanaDeslizante.ipynb**: en este notebook se definen las técnicas clásicas para la detección de objetos en imágenes como la ventana deslizante, la pirámide de imágenes y el *non maximmun supression*. Además contamos con un clasificador (previamente entrenado) que nos dirá la clase del objeto y utilizaremos las técnicas anteriores para realizar una predicción.
- **CPUs-GPUs.ipynb**: en este notebook se ha realizado una comparación de los entrenamientos de varias redes en distintos entornos, en este caso se realizarán las pruebas con la CPU.
- **CPUs-GPUsDrive-GPU.ipynb**: en este notebook se ha realizado una comparación de los entrenamientos de varias redes en distintos entornos, en este caso se realizarán las pruebas con la GPU disponible en Google Drive.
- **YOLO1.ipynb**: este notebook permite descargar la red neuronal YOLO, además de copiar dentro de la carpeta donde se encuentra YOLO un conjunto de notebooks con ejemplos de modelos realizados.
- **YOLO2.ipynb**: este notebook contiene las explicaciones para comenzar a usar YOLO y también contiene una serie de ejemplos donde se puede observar como funciona, tanto para el caso de imágenes como de vídeos.
- **YOLO_PASCAL_VOC.ipynb**: en este caso se ha desarrollado un notebook donde vamos a ver cómo se entrena la red YOLO desde cero utilizando un dataset llamado Pascal VOC.
- **YOLO_CoCo.ipynb**: en este notebook se muestra cómo se entrena la red YOLO desde cero utilizando un dataset llamado CoCo.
- **YOLO_Estomas.ipynb**: en este notebook se muestra cómo se entrena la red YOLO desde cero utilizando un dataset de estomas, el fin de este dataset consiste en dadas unas imágenes que detecte si hay estomas y dónde.
- **CLODSA_Estomas.ipynb**: este notebook nos permite aumentar el dataset en caso de contar con un número reducido de imágenes utilizando la librería CLoDSA. Se muestra cómo hacerlo utilizando el dataset de estomas.
- **NotebookGeneral.ipynb**: en este caso se ha realizado un notebook general que permita crear un modelo de detección, para que cualquier usuario pueda usarlo dado un conjunto de datos cualquiera.

Anexo 2: Tipos de aprendizaje automático

En este anexo se explican brevemente los distintos tipos de algoritmos de aprendizaje automático. Para más información ver [1].

Aprendizaje no supervisado

En los problemas de aprendizaje no supervisado, el algoritmo es entrenado usando un conjunto de datos que no tiene ninguna etiqueta, en este caso, nunca se le dice al algoritmo lo que representan los datos. La idea es que el algoritmo pueda encontrar por sí solo patrones que ayuden a entender el conjunto de datos. El aprendizaje no supervisado es similar al método que utilizamos para aprender a hablar cuando somos bebés; en un principio escuchamos hablar a nuestros padres y no entendemos nada; pero a medida que vamos escuchando miles de conversaciones, nuestro cerebro comenzará a formar un modelo sobre cómo funciona el lenguaje y comenzaremos a reconocer patrones y a esperar ciertos sonidos.

Aprendizaje semi-supervisado

Este tipo de algoritmos combina el aprendizaje supervisado y no supervisado para poder clasificar de manera adecuada. Se tiene en cuenta los datos marcados y los no marcados.

Aprendizaje por refuerzo

En los problemas de aprendizaje por refuerzo el algoritmo aprende observando el mundo que le rodea. Su información de entrada es el feedback o retroalimentación que obtiene del mundo exterior como respuesta a sus acciones. Por lo tanto, el sistema aprende a base de ensayo-error. Un buen ejemplo de este tipo de aprendizaje lo podemos encontrar en los juegos, donde vamos probando nuevas estrategias y vamos seleccionando y perfeccionando aquellas que nos ayudan a ganar el juego. A medida que vamos adquiriendo más práctica, el efecto acumulativo del refuerzo a nuestras acciones victoriosas terminará creando una estrategia ganadora.

Transducción

Similar al aprendizaje supervisado, pero no construye una función de forma explícita. Trata de predecir las categorías de los futuros ejemplos basándose en los ejemplos de entrada, sus respectivas categorías y los ejemplos nuevos al sistema.

Aprendizaje multitarea

Métodos de aprendizaje que usan conocimientos previamente aprendidos por el sistema, de cara a enfrentarse a problemas parecidos a los ya vistos.

Anexo 3: Comparando el uso de CPUs y GPUs en redes neuronales

En el campo de las redes neuronales, y en concreto el deep learning, ha habido una gran revolución gracias al uso de GPUs, ya que ha permitido entrenar redes con una gran cantidad de nodos y con una profundidad considerable en un tiempo razonable. En esta sección se va a realizar una comparativa del tiempo que es necesario para entrenar distintos tipos de redes, en particular LeNet [2], GoogLeNet [3], VGGNet [4], ResNet [5] y ShallowNet [6], en diferentes entornos y ver si es factible trabajar utilizando un ordenador normal sin GPU.

Redes	Número de parámetros	Número de capas
LeNet	1,632,080	12
GoogLeNet	1,656,250	75
VGGNet	2,171,178	23
ResNet	886,102	284
ShallowNet	328,586	5

Tabla 1. Información de las redes.

Está sección va acompañada de un notebook de Jupyter en Python donde se ve como se ha realizado el entrenamiento de dichas redes y predicción de las redes usando para ello el dataset de Cifar [7], que consta de 60000 imágenes, y sirve para construir modelos de clasificación de imágenes en 10 categorías. Dicho notebook es el que se ha ido ejecutando en los siguientes entornos:

- Máquina virtual de Ubuntu.
- Google Drive utilizando la aplicación Colaboratory que da acceso a un ordenador con un procesador Intel(R) Xeon(R) CPU @ 2.30GHz.
- Google Drive utilizando la aplicación Colaboratory que da acceso a un ordenador con una gráfica Nvidia Tesla K80.
- En un servidor con la gráfica Nvidia Titan XP.

	Núcleos	base clock	Max Boost Clock	Memoria	Ancho de banda de la memoria	Ancho de la interfaz de memoria	Tipo de bus
Nvidia Tesla K80	4992	560	875	24 GB	480 GB/s	384 bits	GDDR5
Nvidia Titan XP	3840	1405	1582	12 GB	547,7 Gb/s	384 bits	GDDR5 X

Tabla 2. Comparativa de la gráfica Nvidia Tesla K80 y la Nvidia Titan XP.

Comparación de tiempos de una época de entrenamiento (notar que en aplicaciones reales, las redes se entrenan por cientos o miles de épocas) para las distintas redes en los distintos entornos:

	Máquina virtual	Drive	Drive - GPU	Servidor
LeNet	4 min 31 s	5 min 2 s	10,2 s	7 s
GoogLeNet	1 h 44 min 3 s	2 h 8 min 22 s	1 min 49 s	31 s
VGGNet	17 min 47 s	21 min 43 s	37,8 s	15 s 30 ms
ResNet	3 h 36 min	4 h 31 min 15 s	4 min 26 s	1 min 13 s
ShallowNet	1 min 34 s	1 min 30 s	9,74 s	7 s 39 ms

Comparación de tiempos de predicción, una vez entrenado los modelos, con un total de 10000 imágenes:

	Máquina virtual	Drive	Drive - GPU	Servidor
LeNet	17,5 s	24,5 s	8,5 s	8 s
GoogLeNet	6 min 36 s	7 min 53 s	3 s 48 ms	3 s
VGGNet	1 min 6 s	1 min 23 s	1s 54 ms	1 s 6 ms
ResNet	15 min 10 s	19 min 12 s	7 s 5 ms	8s 15 ms
ShallowNet	7,5 s	8,58 s	1 s 63 ms	2 s 10 ms

Se ve que hay una gran diferencia entre el uso de ordenadores normales contra los que tienen GPU. Si nos fijamos en la comparación de tiempos durante el entrenamiento vemos que en el caso de redes como GoogLenet o ResNet se nota que hay una gran diferencia, hemos pasado de que el entrenamiento dure horas a poder realizarlo en un pocos minutos o segundos en el caso de servidor. El uso de gráficas nos permite avanzar con mayor rapidez, por lo que si disponemos de una gráfica es recomendable usarla frente a ordenadores convencionales ya que permite hacer pruebas más rápido y comprobar si las redes están aprendiendo correctamente. Esta comparativa demuestra que no es factible trabajar con una CPU cuando se quiere entrenar una red profunda ya que le cuesta demasiado procesar los datos.

Anexo 4: Tipos de arquitecturas

R-CNN

Actualmente, las redes convolucionales han demostrado ser lo suficientemente importantes,
tanto, que aplican mejoras en la clasificación de imágenes así como en la detección de objetos. Eso sí, si comparamos la clasificación con la detección, ésta última ofrece una mayor dificultad a la hora de su realización, lo que conlleva la utilización de detectores mucho más complejos que los utilizados anteriormente en otros tipos de detectores.
El detector R-CNN [8], está dividido en 4 etapas. La primera etapa es la imagen de entrada al sistema. En la segunda etapa se generan las propuestas a regiones de interés utilizando la Búsqueda selectiva [9]. El tercer módulo consiste en una gran red neuronal convolucional que extrae un vector de características de longitud fija para cada región. Y por último, la cuarta etapa consiste en un conjunto de clases lineales SVM. La figura 1, muestra estas etapas.

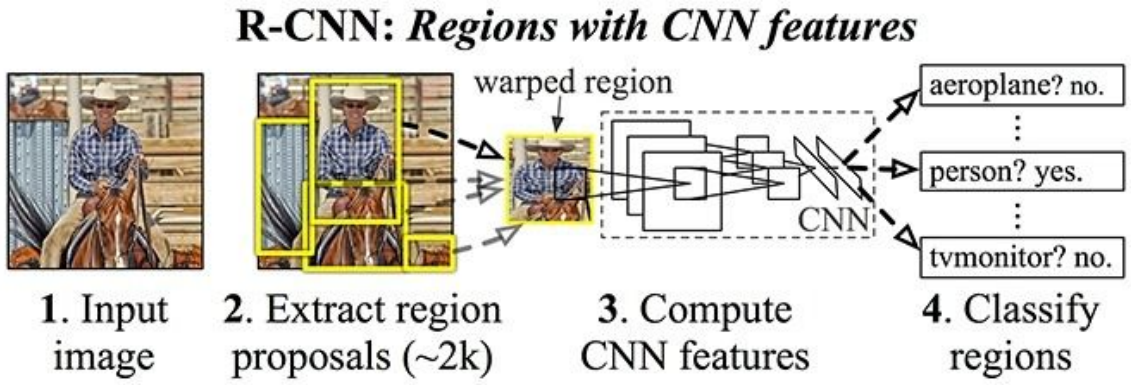


Figura 1. Arquitectura R-CNN.

Aunque se pueden obtener grandes resultados, el entrenamiento tiene muchos problemas. Sin embargo, este algoritmo de detección conlleva un alto coste computacional lo que lo hace extremadamente lento.

Fast R-CNN

Este enfoque de detección de objetos evolucionó rápidamente hacia un aprendizaje profundo más puro, un año después de R-CNN se publicó Fast R-CNN [10] que consigue solventar los problemas que ocasionan la utilización de dichas redes convolucionales.

En la figura 2, se puede apreciar la arquitectura que sigue el detector FAST R-CNN. En un primer paso, este detector toma una imagen de entrada, I, y una serie de regiones de interés (usando para ello la búsqueda selectiva). Posteriormente, el detector introduce a I y

a dichas regiones a una red convolucional y las agrupa en capas de agrupación máximas para crear el mapa de las características de convoluciones. Luego, para cada propuesta de región de interés, obtenidas en el primer paso, se le extraerán vectores de características de tamaño fijo procedentes del mapa de características. Cada vector es añadido a continuación de otros para crear una secuencia totalmente conectada, FC, que finalmente se divide en dos capas iguales. Una de ellas produce estimaciones de probabilidades *softmax* sobre K clases de objetos más una clase de tipo fondo. La otra de las capas genera cuatro números de valores reales para cada una de las K clases de objetos. Cada conjunto de 4 valores codifica las posiciones de la caja delimitadora para una de las clases K. Todo este proceso viene mucho más desarrollado en el artículo [10].

Este enfoque no solo fue más rápido, sino que el modelo fue más fácil de entrenar. El mayor inconveniente fue que el modelo aún dependía de la Búsqueda selectiva (o de cualquier otro algoritmo de propuesta de región), que se convirtió en el cuello de botella.

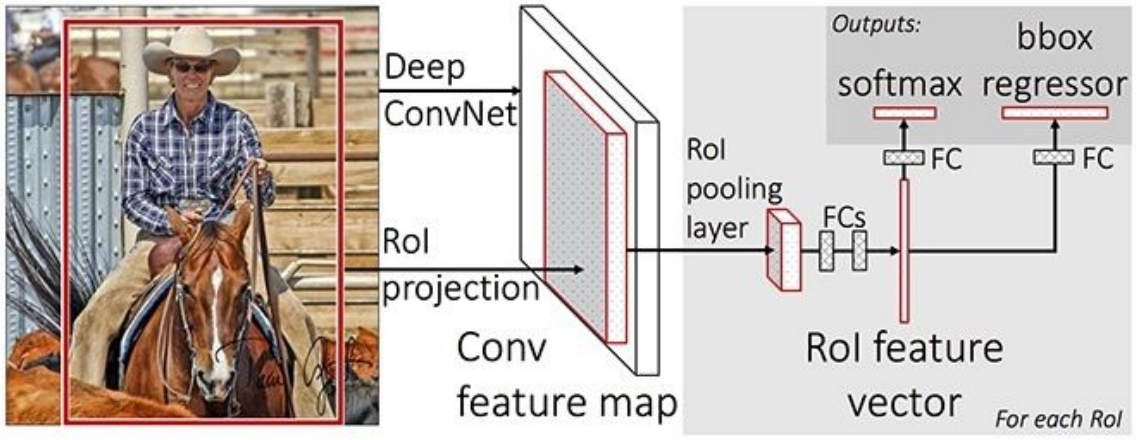


Figura 2. Arquitectura Fast R-CNN.

Faster R-CNN

Posteriormente, Faster R-CNN [11] es la tercera versión de la serie R-CNN. Faster R-CNN agregó lo que llamaron una Red de Propuestas de Región (RPN), en un intento de deshacerse del algoritmo de Búsqueda Selectiva y hacer que el modelo sea completamente entrenable de punta a punta. El autor propone un sistema de detección de objetos formado por dos módulos. Un primer módulo compuesto por una red completamente convolucional, cuya finalidad es la de proponer las regiones de interés; más otro segundo módulo coincidente con el detector FAST R-CNN que use esas regiones de interés propuestas. La figura 3, muestra estas dos etapas que se pueden considerar como una sola red unificada para la detección de objetos, vamos a explicarlas un poco.

El primer módulo, al que podemos llamar *Region Proposal Network* (RPN) [12], toma cualquier imagen como entrada y devuelve un conjunto de rectángulos de las zonas de interés cada uno de ellos con una puntuación objetiva. Para generar las regiones, al mapa de

características convolucionales se le aplican una serie de pequeñas redes mediante ventanas deslizantes. A cada una de estas ventanas deslizantes se le asignará una característica de menor dimensión. Ésta alimentará a dos capas hermanas conectadas, una capa con la clasificación de la caja, cls, y otra con las regresiones de caja, reg. Simultáneamente, para cada una de las localizaciones que tome la ventana deslizante se calcularán las posibles regiones de interés siendo el máximo número de regiones k. Por lo tanto, la capa reg contendrá las coordenadas de las posibles regiones, mientras que la capa cls contendrá las probabilidades de que dichas regiones sean objeto o no.

Una vez calculadas estas regiones de interés se pasará a la detección mediante el algoritmo FAST R-CNN. Por lo tanto, el algoritmo FASTER R-CNN está compuesto por RPN más FAST R-CNN, siendo este primero el que indica donde buscar al segundo.

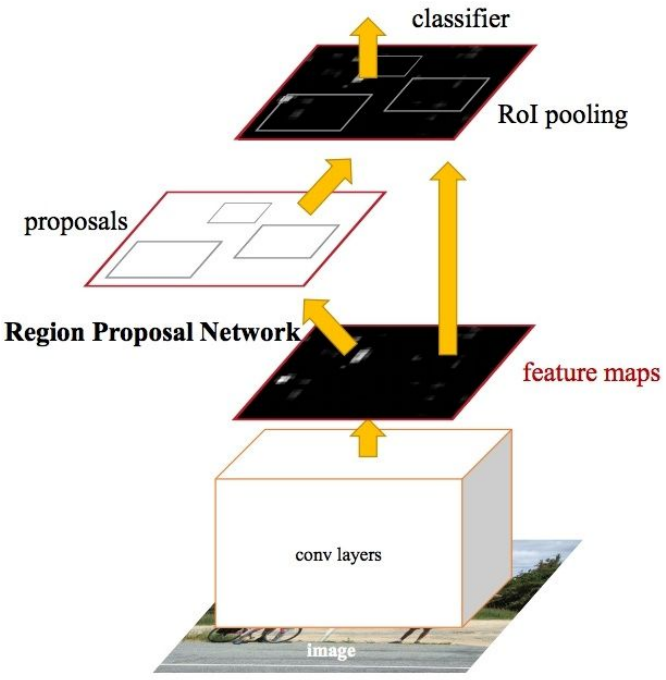


Figura 3. Arquitectura FASTER R-CNN.

SSD

Finalmente, hay dos documentos notables, Single Shot Detector (SSD) [13] que toma YOLO utilizando mapas de características convolucionales de múltiples tamaños para lograr mejores resultados y velocidad, y Redes completamente convolucionales basadas en la región (R-FCN) que toma la arquitectura de FAST R-CNN pero con solo redes convolucionales.

El detector SSD difiere de otros detectores de disparo único debido al uso de múltiples capas que proporcionan una precisión más fina en objetos con diferentes escalas. (Cada capa más profunda verá objetos más grandes).

La SSD normalmente comienza con un modelo pregenerado de VGG [14] en Resnet que se convierte en una red neuronal completamente convolucional. Luego, agregamos capas de conv.as adicionales, que en realidad ayudarán a manejar objetos más grandes. La arquitectura SSD puede, en principio, usarse con cualquier modelo de base de red profunda.

Un punto importante a tener en cuenta es que una vez que la imagen se transfiere a la red VGG, se agregan algunas capas conv que producen mapas de características de tamaños 19x19, 10x10, 5x5, 3x3, 1x1. Estos, junto con el mapa de características 38x38 producido por conv4_3 de VGG, son los mapas de características que se usarán para predecir recuadros delimitadores.

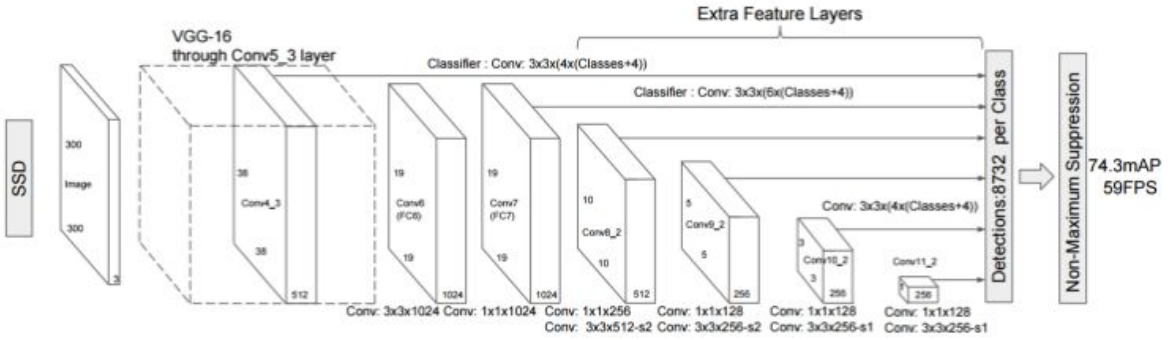


Figura 4. Arquitectura SSD.

Referencias

1. González A. Conceptos básicos de Machine Learning - [Internet]. 30 Jul 2014 [cited 21 Jun 2018]. Available: http://cleverdata.io/conceptos-basicos-machine-learning/

2. Lecun Y, Bottou L, Bengio Y, Haffner P. Gradient-based learning applied to document recognition. Proc IEEE. 1998;86: 2278–2324.

3. Szegedy C, Liu W, Jia Y, Sermanet P, Reed S, Anguelov D, et al. Going deeper with convolutions. 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). 2015. doi:10.1109/cvpr.2015.7298594

4. Applications - Keras Documentation [Internet]. [cited 11 May 2018]. Available: https://keras.io/applications/#vgg16

5. He K, Zhang X, Ren S, Sun J. Deep Residual Learning for Image Recognition. 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). 2016. doi:10.1109/cvpr.2016.90

6. Ricky TR. Shallow Neural Network from scratch (deeplearning.ai assignment). In: Towards Data Science [Internet]. Towards Data Science; 20 Dec 2017 [cited 11 May 2018]. Available: https://towardsdatascience.com/shallow-neural-network-from-scratch-deeplearning-ai-assignment-320f57c581cd

7. CIFAR-10 and CIFAR-100 datasets [Internet]. [cited 11 May 2018]. Available: https://www.cs.toronto.edu/~kriz/cifar.html

8. Girshick R, Donahue J, Darrell T, Malik J. Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation. Proceedings of the 2014 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'14). 2014. pp. 580–587.

9. Uijlings JRR, van de Sande KEA, Gevers T, Smeulders AWM. Selective Search for Object Recognition. Int J Comput Vis. Springer US; 2013;104: 154–171.

10. Girshick R. Fast R-CNN. Proceedings of the 2015 IEEE International Conference on Computer Vision (ICCV 2015). 2015. pp. 1080–1088.

11. Ren S, He K, Girshick R, Sun J. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. IEEE Trans Pattern Anal Mach Intell. 2017;39: 1137–1149.

12. How does the region proposal network (RPN) in Faster R-CNN work? In: Quora [Internet]. [cited 21 Jun 2018]. Available: https://www.quora.com/How-does-the-region-proposal-network-RPN-in-Faster-R-CNN-work

13. Liu W, Others. SSD: Single Shot MultiBox Detector. Proceedings of the 14th European Conference on Computer Vision (ECCV 2016). 2016. pp. 21–37.

14. Applications - Keras Documentation [Internet]. [cited 21 Jun 2018]. Available: https://keras.io/applications/#vgg16